Section 18
Flight Table Manager Function

The purpose of the *Flight Table Manager (FTM)* function is to maintain a database containing flight information on all flights operating under the FAA Instrument Flight Rules (IFR) and to provide timely updates of aircraft positions to the *Aircraft Situation Display(ASD)* function. The *FTM* also provides flight data to various programs upon request.

Design Issue: Database Design

The information maintained in the *FTM* database includes aircraft flight plans, flight plan modifications, position reports, arrival and departure data, and route data required to produce map files.

The *FTM* database is composed of seven *shared regions* and one header file. These regions reside in the **/ftm_check** directory on the node running the *FTM* process. Other processes running on this node may access these regions, but only the *FTM* process is authorized to write to any of them. Each region is named **ftm_checkpoint_file_x**, where **x** is a letter from **a** to **g**.

The file **ftm_checkpoint_file_a** contains the often used portion of each flight record. Since most of the data correlation of messages to *FTM* can be performed with this data, it was intentionally designed for its contents, size, and page alignment properties for fast and frequent traversal. The key to this region is named **flight_array1**.

The file **ftm_checkpoint_file_b** contains the variable portion of the flight record. This includes waypoints (4 bytes each), sectors (6 bytes), fixes (6 bytes), airways (6 bytes), Air Route Traffic Control Centers (ARTCCs, 3 bytes), and field 10 messages (1 byte) generated by the Flight Database (FDB). This data is stored adjacent to each other with no intervening spaces. The *FTM* maintains the number of entries in each field and upon extraction moves the appropriate number of bytes from the offset according to the following formula:

waypoints*4 + sectors*6 + fixes*6 + airways*6 + ARTCCs*3 + route_size

The route data is stored in 248 byte pages, with an additional eight bytes used for addressing information. Thus, 518 bytes of data would occupy three pages with the third only using twenty-two of the available bytes. The format of the eight control bytes is

- 1 through 4 – Address of the flight record owning this page
- 5 through 6 – Size code (32000 = continuation) the number of pages used
- 7 through 8 – Number of bytes used on this page

The key to this file is the record **flight_array_rte**.

The file **ftm_checkpoint_file_c** contains the flight hash table. The aircraft identifier (**ACID**) is hashed to a location in this table. That location provides an index into the **ftm_checkpoint_file_a** file which begins the linked list. If the first record is not the desired **ACID** the Next One field is checked to provide the location of the next element in the list. The key to this file is **flight_table_hash**.

The file **ftm_checkpoint_file_d** contains the flight table **in use** bitmap. If the value is one then that position is in use within files **ftm_checkpoint_file_a** and **ftm_checkpoint_file_g**. If the value is zero then that subscript position is available. The key to this file is **flight_storage**.

The file **ftm_checkpoint_file_e** contains the active table. This table is used to facilitate map making. The file contains an active hash table, an **in use** bitmap, and a linked list. Each element in the list contains an **ACID**, index into **ftm_checkpoint_file_a**, and a pointer to the next element in the list. The key to this file is **flight_active**.

The file **ftm_checkpoint_file_f** contains the airport table. This table lists all flights arriving at and departing from specified airports (supplied by **pacing.dat** file) for twelve hours in the past to twelve hours in the future. Each element in the table contains the airport identifier, time, arrival time, and departure time with an index into **ftm_checkpoint_file_a**. The key to this region is **flight_airport**.

The file **ftm_checkpoint_file_g** contains the low-use fixed portion of the flight record. There is a one-to-one correspondence between this file and **ftm_checkpoint_file_a**. A record in this file contains a pointer to the location of the variable portion of the flight record in **ftm_checkpoint_file_b**. The key to this file is **flight_array2**.

The file **shared_region_header** is a record file containing information about the shared regions. This information is maintained to help the *FTM* determine the validity of the database upon startup. Fields of this record include the region creation time, the region running time, the number of times *FTM* has been started and stopped since the region was created, and the total number of flights created and deleted since the region was created. The key to this file is **shared_region_header**.

In summary, a flight record in the *FTM* database is found by hashing the aircraft identifier (up to 7 characters), which produces an index. This index is then used to traverse the linked list found in **ftm_checkpoint_file_a** until the appropriate entry is located. The ordinal position of this entry in **flight_array1** is used to obtain the remainder of the fixed portion of the flight record in **flight_array2**. This array contains the address of the first page containing the variable sized route information (in **ftm_checkpoint_file_b**) under the pointer **flight_array_rte**.

When a new flight record is to be added, the next available position is found by incrementing the global variable **flight_array_last_slot**. If this variable is greater than **total_flight_records** it is reset to one. If the **in use** bit in **ftm_checkpoint_d** is set, then the procedure is repeated until an available position is found. When this occurs, that address is used for the new data, and the bit is set to **in use.**

To add a record to the active table (**ftm_checkpoint_file_e**) the next available position is found by incrementing the variable **flight_active_last_slot**. If the value exceeds **number_actives_allowed**, the last slot is reset to one. The active **in use** bit map is searched until an available position is found. Then that address is used and the bit is set to **in use.**

When there is route data to be added, the number of pages (248 bytes each) required is calculated. If there is a route page area already allocated for this flight and the page count is the same, the old data is overwritten. Otherwise, a suitable space must be found. To find the available space the variable **flight_array_rte_last_slot** is incremented. If it exceeds **total_rte_pages**, it is reset to one. The bitmap is checked to see if there is the correct number of adjacent pages. If not, the procedure is repeated until an appropriate location is found. When found, the **in use** bit is set in the RTE portion of **flight_storage**.

NOTE: When the *FTM* is restarted, the shared regions are reloaded with previous values, and all last slot variables are reinitialized to one.

Processing Overview

Figure 18-1 illustrates the data flow of *FTM* functions at the ETMS hub site; the normal mode is shown. For slave mode, substitute FDBD with Master FTM. For backup mode, substitute FDBD with Master FTM, change FTM to FTMB, and remove clients. Client type is shown in parenthesis. For further discussion of the different modes, refer to Section 18.1.

Figure 18-2 depicts the data flow of the *FTM* function at each field site. The *FTM* extracts appropriate information received from the *Flight Database Distributor (FDBD)* and adds it to its flight tables, and every map cycle interval it produces a report of all active flights, which it distributes to the *ASD* in the form of map and route (**rte**) files. The field site *FTM* also processes requests from the *ASD* for specific flight information.

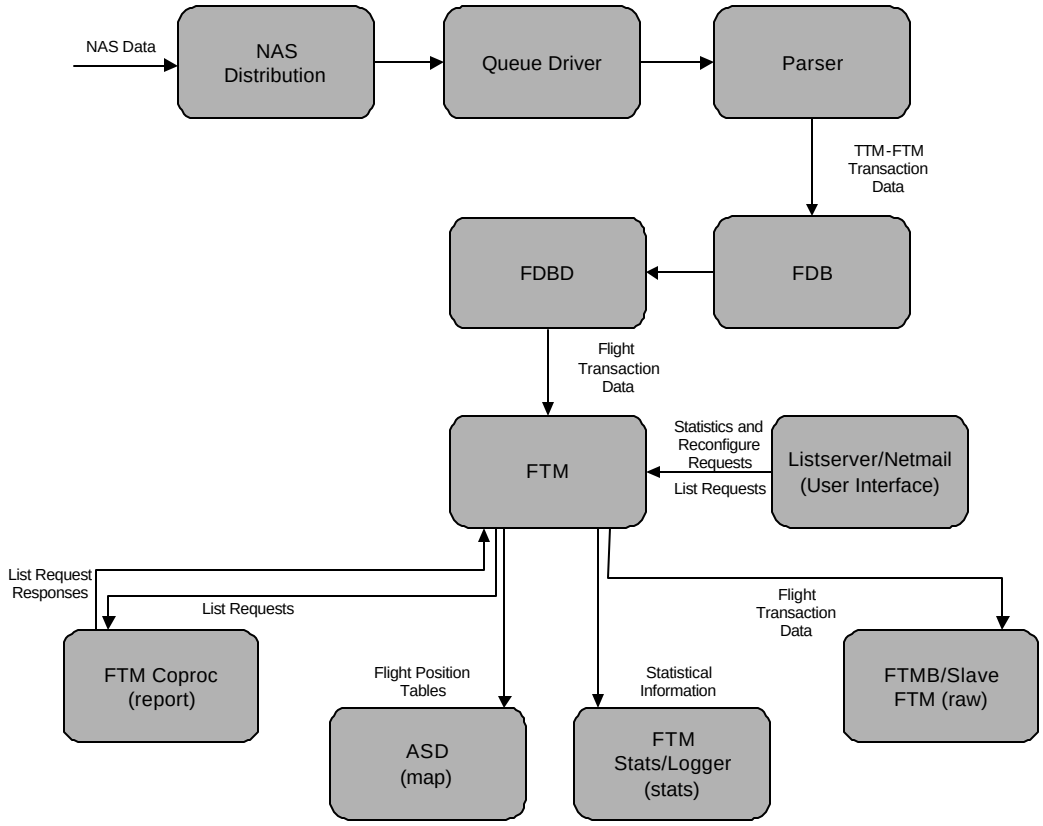

Figure 8-1. FTM Interface at Central Site

The *FTM* Function is composed of two processes: *Flight Table Manager (FTM)* and *FTM_Coprocess*, as shown in Figure 18-3. The *FTM* process requests for and receives data from *FDBD*, processes the data, maintains the *FTM* database, produces map files, and returns replies. The *FTM_Coprocess* program receives requests for reports from the *FTM*, processes them, and forwards the reply to the requesting process. Both programs have *read* access to the shared regions, but only the *FTM* process has *write* access. If the *FTM_Coprocess* program is not available, the *FTM* process will produce the desired reports.

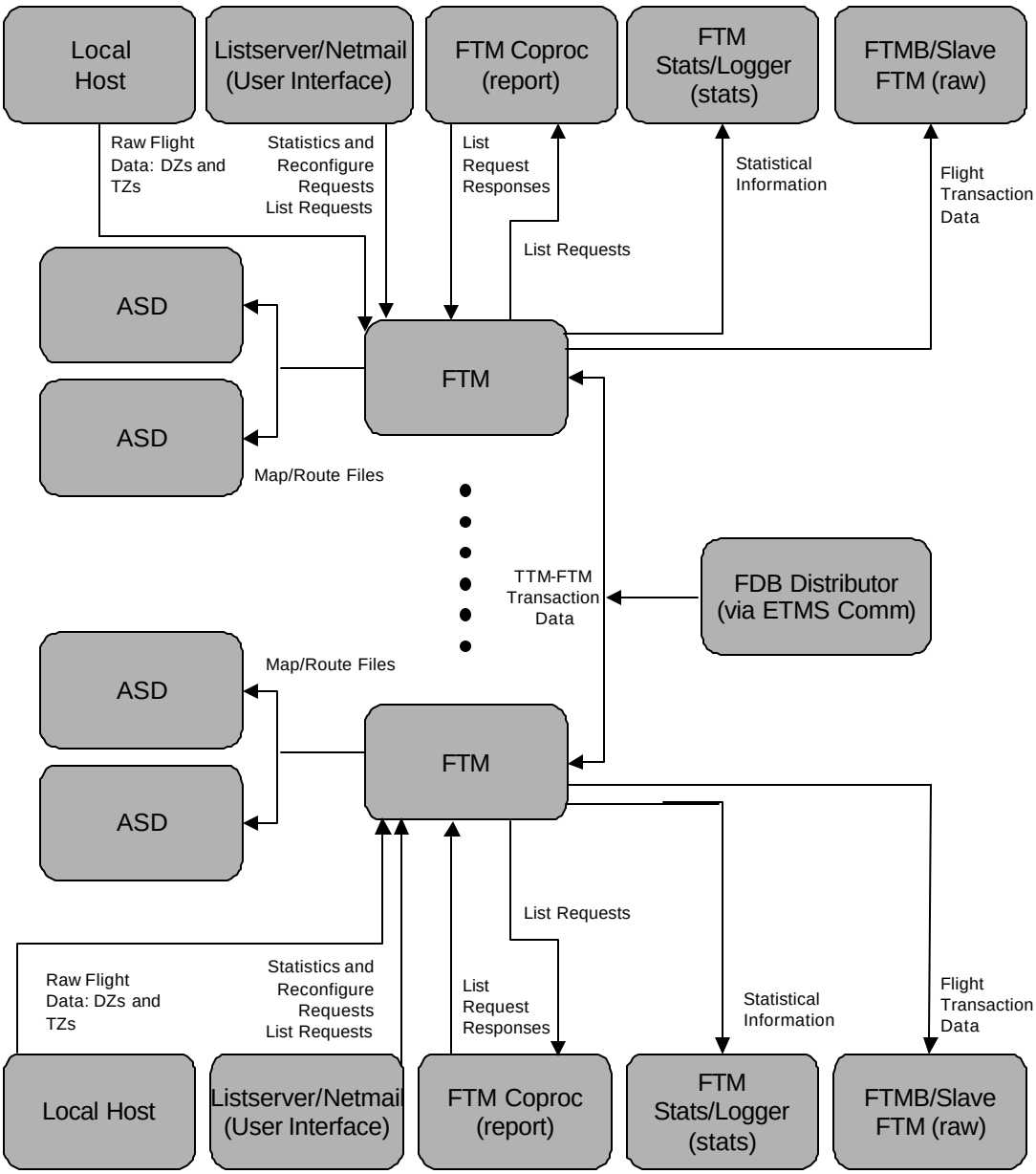

Figure 18-2. FTM Interface at Field Sites

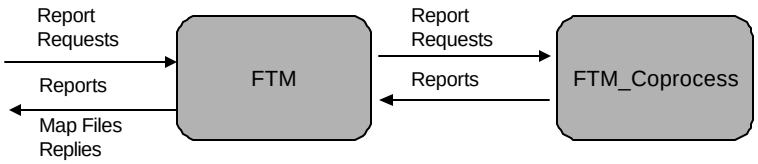

Figure 18-3. Data Flow of the FTM Process

18.1 Flight Table Manager Process

Purpose

The main purpose of the *Flight Table Manager* (*FTM*) is to maintain the *FTM* database, which is built from flight data messages (transaction messages) received from the *FDBD* process. The *FTM* uses this information to determine aircraft profiles and positions. The *FTM* produces and distributes a report (map file) of all flights currently aloft (termed *active*) every minute (the number of minutes between reports is user-configurable, see next section) to all map clients, i.e., the *ASD* program. These reports are used by the *ASD* to display the aircraft positions. The *FTM* also processes requests from the *ASD* for specific flight information.

Other functions performed by the *FTM* include ETMS Network Address interfacing, statistics collection, user request processing, reconfigure processing, recovery data processing, raw data parsing, backup FTM *(FTMB)* implementation, error-logging, and disk archiving of data.

Modes. The *FTM* operates under one of three modes, normal mode, slave mode (also known as tail mode), and backup mode. The mode determines the method of obtaining flight transaction data generated by the FDBD process at the hub site. Transaction data is the record-formatted flight information which has been fully processed by the ETMS Version 5 software, i.e. Parser and FDB.

Under normal mode, *FTM* registers to receive data directly from an FDBD process. The operational FDBDs run at the hub site.

Under slave mode, the *FTM* registers to receive data from another *FTM*, usually at a different field site. Slave *FTM*s do not request recoveries, allowing their master *FTM* to manage the FDBD connection.

Under backup mode, the *FTM* registers to receive data from another *FTM*, usually at the same site. Functionally, backup *FTM*s operate differently from normal and slave *FTM*s in the following ways: Backup *FTM*s do not generate map files if an output directory is unspecified in the configuration file, do not request recoveries, do not accept clients, and connect to Network Addressing as class *FTMB*, as opposed to *FTM*.

The mode is specified in the *FTM* configuration file, which is passed as an argument and processed at startup. *FTM* can read and process a new or modified configuration file, changing modes if specified, in response to Net.Mail *reconfigure* commands. The reconfigure command may be used in either of the methods described below:

<reconfigure> < address of the FTM process> [configuration filename]

NOTE: If a configuration file is not specified, *FTM* re-reads the current file.

<reconfigure> < address of the FTM process> $<primary FDBD site>

NOTE: If a secondary site is not supplied, it is assigned the same as the primary site.

Clients. *FTM* accepts four types of clients: report, map, stats, and raw.

Report clients register to *FTM* to assist in the processing of user-initiated flight database queries. *FTM_Coproc* is a report client. For example, *FTM* receives an F ARR BOS command from a *Net.Mail* process. *FTM* forwards this request to *FTM_Coproc*, which processes the request, and returns the response information to *FTM*. *FTM* then forwards the response to the requestor (*Net.Mail*).

Map clients register to *FTM* to receive notification of new map filenames from *FTM*. *ASD* is a map client which reads these map files, and displays active flights based on their characteristics as described within the map file.

Stats clients register to *FTM* to receive general statistics about the *FTM*, for example, map file time, size, and name. Logger is a stats client.

Raw clients register to *FTM* to receive flight transaction data. Slave and Backup *FTM*s are raw clients. As the master *FTM* receives transaction data, it is passed on to any raw clients.

FTM Configuration File Description. The *FTM* configuration file contains parameter values that define the program's environment. These parameters can be dynamically changed, via the reconfigure command in *Net.mail*, without restarting the program or clearing the *FTM* database.

Each parameter setting uses one line in the configuration file and each parameter has an associated symbol, which must be specified in the first column of the line. Table 18-1 describes the parameters.

Table 18–1. Configuration File Symbols

This parameter symbol in the first column of a line . . .	Instructs *FTM* to . . .
#	Ignore the content that follows. This symbol is intended to indicate comments.
$ followed by a site name	Interpret that the name is the primary data source site. *FTM* looks for an *FDBD* process at the specified site to register as a client. The second line beginning with $ indicates the secondary data source site, which will be switched to in case of a connection (one minute) or data (two minutes) timeout on the primary site connection. This is the normal *FTM* operation.
***** followed by a site name	Run as a backup *FTM* (class *FTMB*), registering to a master *FTM* at the specified site as a raw client. (An *FTMB* process maintains its own flight database and does not create map files or perform recoveries.)
% followed by a site name	Run as normal (class *FTM)*, but to register as a raw client with an *FTM* at the specified site. This type of *FTM* will be referred to as a Tail *FTM.* Tail *FTMs* are intended to be used as a site's sole *FTM*, creating map files, but not performing recoveries (because their master *FTM* will request any necessary recoveries).
! followed by a directory name	Write its **map**, **route**, **orig files** output files on the node. If this symbol is not specified or the directory provided does not exist, *FTM* will use the default of **/traffic**.
^ followed by a digit from 1 to 9	Interpret that this is the number of minutes between each map file creation (map cycle interval). The *FTM* reads only the first character after the ^ and ignores the remainder of the line. Any invalid entry will leave the map cycle interval set to its current value, if one exists, or default to three minutes.
R or r in the first column	Write any raw data received to an hourly log file in the output directory under the name **rawdata.<timestamp>.** The default is not to write these files.
& followed by a digit from 1 to 9	Interpret that this is the number of minutes the *FTM* will allow each ARTCC to go without receiving a Position Update message (TZ) before reporting its flights as ghosted (ghost determination interval). This line is optional, and the default value is based on the map cycle interval below. *FTM* reads only the first character after the **&** and ignores the remainder of the line. Any invalid entry will cause the ghost determination interval to be set to its default. Refer to Note 2 that follows for the ghost determination default.
M or **m**	Turn the military filter on. The default is off.
O or **o**	Write **orig** (**log**) files of flight transaction data received. As a default, *FTM* does not write the **orig** files.

NOTE 1: The configuration file must have one of the $, *, or % specifiers in order to run properly. If more than one of them exist in the file, the order of precedence is *, %, $.

NOTE 2: The ghost determination default is based on the following map cycle interval.

Map Interval	Ghost Interval
1 minute	3 minutes
2 minutes	2 minutes
anything else	1 minute

NOTE 3: *FTMB*s do not respond to the *Net.mail* **f** commands. Use the corresponding *Net.mail* **s** commands to retrieve stats from FTMBs.

Design Issue: Module Design

FTM is composed of the modules illustrated in Figure 18-4.

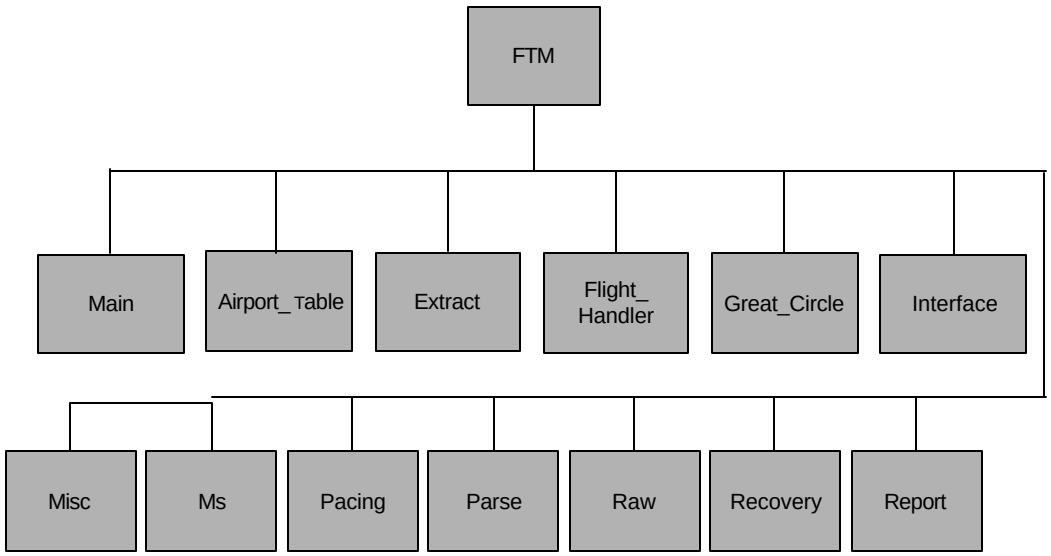

Figure 18-4. FTM Modules

Following is a description of the *FTM* modules :

- *FTM Main* defines global variables, invokes the initialization module, transfers control to the processing routine, and, when termination is signalled, invokes the termination logic.
- *Airport_Table* maintains the airport tables found in the Shared Region **ftm_checkpoint_f**.
- *Extract* is concerned with generating the map files sent to *ASD* every map cycle interval.
- *Flight_Handler* is responsible for adding, modifying, searching, and deleting entries within the FTM flight database (**ftm_checkpoint** files).

- *Great_Circle* is used to calculate the great circle route for international flights. It derives the heading and distance between a source position and its destination. Since it required several trigonometric functions not available in Pascal, it is the only routine coded in Fortran.
- *Interface* provides the interface to the ETMS Version 5 message switching system. This module processes incoming messages, sends outgoing messages, manages the client table, generates statistics and other reports, and registers for flight data.
- *Misc* includes routines providing miscellaneous functions, including program initialization, the main processing loop, queue management, configuration file reading, time synchronization, and error reporting.
- *Ms* handles the flight database creation, mapping, and unmapping (**ftm_checkpoint** files).
- *Pacing* maintains tables that contain the counts of arrivals and departures from pacing airports.
- *Parse* All transaction messages are parsed and processed by the routines in this module. The information is used to update the flight data tables in the database.
- *Raw* parses raw flight data only for Departure messages (DZs) and TZs and incorporates the data into the *FTM* database.
- *Recovery* maintains recovery data statistics and handles the recovery protocol with the *FDB Recovery (FDBR)* process.
- *Report* generates reports in response to user requests, writes them into files, and forwards them to the requesting address. This module is shared by the *FTM_Coprocess*.

Execution Control

The *FTM* process is normally started by the *Nodescan* utility after a new software release or after a software or hardware crash.

Input

FTM has one mandatory argument and one optional argument at startup:

- The FTM configuration filename is a required argument. If a valid configuration file is not supplied, the program will terminate. See the above section *FTM Configuration File Description* for detailed information.
- The optional argument **-path <etms_path>** may also be supplied, indicating the path where program-related objects, such as the ftm trace file, may be found. If this argument is not supplied, no path will be prepended to these objects.

The following is a list of the types of *FTM* input:

- Packed transaction messages from the *FDBD*

- Raw National Airspace System (NAS) messages from the local site
- Requests for data from the *ASD* or other programs in the ETMS
- Requests from the *Net.mail* program to reconfigure
- Requests from the *Net.mail* program for statistics
- Registration requests from clients
- User requests for specific flight data
- Remarks keywords file for National Route Program (NRP) parameters.

The input directory specified in the configuration file must contain the following three files which are necessary to execute the *FTM* process:

- **/etms5/ftm/data/airstrip.dat –** The airport file contains the airport identifiers in alphabetical order and the latitude and longitude of each airport. The numeric fields are floating point ASCII, the integer portion indicating the degrees and the decimal portion being the number of minutes divided by 60. If the airport is below the equator or east in lat/lon, a negative sign is placed in front of each value. Sample entries are
  - **ABQ 3.504175E+01 1.066063E+02**
  - **ABR 4.545000E+01 9.843333E+01**
- **/etms5/ftm/data/ –** The pacing airport file contains an alphabetical list of airport identifiers that the *FTM* process will monitor for the **;A** command and for rapid *ASD* retrieval. It contains one airport code per line. For example:
  - **ATL**
  - **BOS**

All entries in each file must be alphabetical. *FTM* may not function properly if they are not.

Output

The following is a list of the types of *FTM* output:

- Map filenames containing flight data that are sent to the ASD at the time of each map cycle interval
- Statistical information which is sent to Ftm_stats, the statistical display program
- Responses to data registration requests from other processes
- Responses to statistics requests from Net.mail
- Responses to reconfigure requests from Net.mail
- Report requests forwarded to FTM_Coprocess for processing
- Reports resulting from user requests which were not able to be handled by the FTM_Coprocess

The following is a list of the output data files generated by the *FTM*:

- **map** files – contains the most recent flight data for all active flights. These files are created every map cycle interval (usually one minute or three minutes) and reside in the output directory specified in the configuration file (usually **/traffic**).
- **rte** files – contains route information for active flights; has a one-to-one correspondence to the map files. These files are created every map cycle interval (usually one minute or three minutes) and reside in the output directory specified in the configuration file (usually **/traffic**).
- **ftm_trace_log** – contains the trace-back information that identifies the status of the *FTM* when it last terminated. This file resides in the **/ftm/trace** subdirectory of the ETMS path specified by the program's second argument (usually **/etms5**).
- **orig files** (if turned on) – contains the packed transaction messages (as received from the *FDBD* process). These files are created on the hour and reside in the output directory specified in the configuration file. A separate program *dump_orig* has been written to unpack the messages from these files and write them to the screen.
- Raw data files – an hourly archival of raw messages received.

The files in the directory **/ftm_check** is the core of the *FTM* database. It is a group of files containing the *FTM* database. It is updated dynamically and closed whenever the *FTM* is terminated.

Processing

Flight data, both transaction and raw, comes in the form of various message types. The following NAS message information is processed by the *FTM*:

- FS Flight Schedule
- FZ Flight Plan
- DZ Flight Departure
- TZ Position Report
- TO Oceanic Position Report
- AF Flight Amendment
- UZ ARTCC Boundary Crossing Notification
- AZ Flight Arrival
- RZ Flight Cancellation
- RS FS Cancellation
- EDCT Flight under Controlled Time
- 5-SETBACK Five-Minute Flight Departure Delay

- SI-CANCEL Flight Cancellation due to Substitution
- CTL-CANCEL Cancellation of Flight under Controlled Time
- BLOCK ALT. Altitude Range Specification
- CRITICAL Critical Recovery Information
- TTM_FTM Remaining Recovery Information
- *RAW_TZ Raw TZ message
- *RAW_DZ Raw DZ message

NOTE: All flight messages above are transaction type messages except those denoted by *, which are raw messages.

18.1.1 The FTM Main Module

Purpose

This module contains the main section of the *FTM* process. It defines the global variables, establishes constants, and passes control to other modules.

Input

None.

Output

The following global variables:

- status
- etms_valid
- timer
- cleanup_handler

Processing

Figure 18-5 illustrates the logical flow for the *Main* program. If no faults are detected upon program initiation by the fault handler, the *Set_Timer* routine (*Misc* module) is called. The **start_time** variable is set, the *Initialize* routine (*Misc* module) is called, the program timers are set and the *Process* routine (*Misc* module) is called. Otherwise, if a fault is detected, the error handling routines (*Trace_Back* [*Misc* module] and *Flight_Table_Unmap* [*Flight_Handler* Module]) are called, all open streams are closed, all clients are closed (*Close_Clients* routine [*Interface* module]) and registration to the data provider is cancelled (*Register_To_Provider* [*Interface* module]).

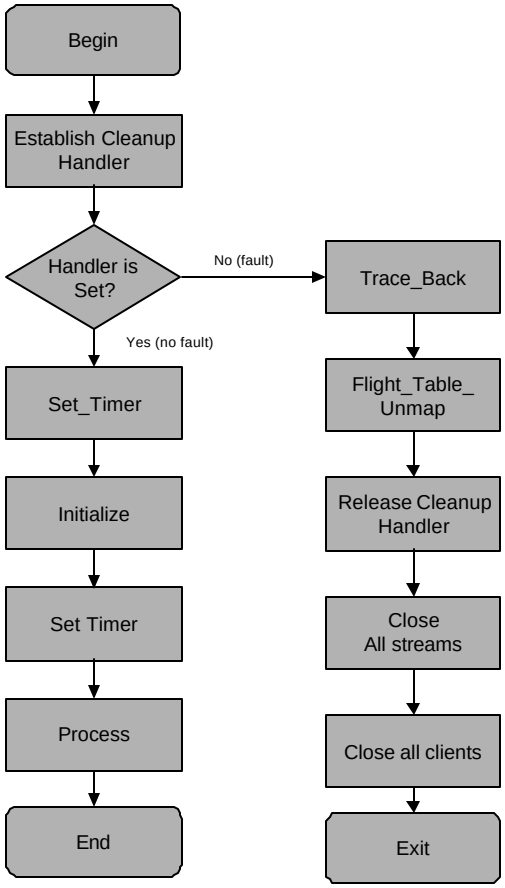

Figure 18-5. Sequential Logic for the FTM Main Program

Error Conditions and Handling

The *FTM* Main Module calls the program cleanup handler upon program termination. Specific error messages are listed in Table 18-21.

18.1.2 The Airport_Table Module

Purpose

This module maintains the airport table found in the Shared Region **ftm_checkpoint_file_f** and the pacing table, which is held in memory.

Input

A *Shared Region* file– **ftm_checkpoint_file_f.**

Output

Changes in the input file.

Processing

The routines in the *Airport_Table* module are concerned with maintaining the airport table and the pacing table. The airport table contains aircraft identifiers that are proposed or have arrived or departed in each 15-minute interval for 12 hours past and 12 hours future. The pacing table contains the number of arrivals and departures for each 15-minute interval (bucket) for each pacing airport.

Routines. The *Airport_Table* module is composed of the following independent *routines* that run when called:

- *Airport_Get_Position* is given an airport identifier and a time. It returns the index into the airport table and the appropriate time slot. It calls *Find_Airport_In_Pacing* to find the airport identifier in the list. It then uses ***Flight_Airport^.which_is_first*** and ***Flight_Airport^.Interval_start*** to search the circular buffer and determine the correct time slot.

- *Airport_Header_Add_Flight* gets a flight pointer, airport identifier, time, and whether it's an arrival or a departure. First, this routine calls *Airport_Get_Position*, which validates the airport identifier and returns the correct time slot. If either of these values is invalid, the routine will return. If the flight is already in that time bucket, the routine will return. Otherwise, it will increment the bucket count for arrival or departure. If it exceeds the maximum, it will set it to the maximum and add the flight data, where the bucket count indicates; thus, having excess flights during a 15-minute interval causes overwriting.

- *Airport_Header_Advance_Interval* deletes intervals that are too old and ensures that the first interval is the correct one. First, it determines what the first interval start time should be. Then, it advances until it reaches that interval. As it advances, it zeroes out the arrival and departure counters and increments the variable ***Flight_Airport^.which_is_first***. If the end of the buffer is reached, it wraps around to the beginning.

- *Airport_Header_Create_Table* reads the airport codes from **pacing.dat** into a table. Then it determines the time of the first 15-minute interval and clears out all counters for all intervals for each airport in the table.

- *Airport_Header_Delete_Flight* uses a boolean value to determine if the flight is an arrival or departure flight. Based on this and a time code passed in, it sets a time value. It calls *Airport_Get_Position* to retrieve the index and the time slot. Finally, it goes through the table to find the flight. When found, all the following entries are moved down in the table, and the arrival or departure counter is decremented.

Error Conditions and Handling

If there is an error opening the **pacing.dat** file, a call is made to Error_$print. If there is an error reading a record from this file, a message is written to the trace log, and that record is skipped. Specific error messages are listed in Table 18-21.

18.1.3 The Extract Module

Purpose

The *Extract* module extracts the flight information for every active flight in the *FTM* flight table database, and generates the map files. The map files contain the information that the *ASD* program uses to produce the flight display.

Input

A *Shared Region* file – **ftm_checkpoint_file_e**, which holds the active flight table**.**

Output

The *Extract* module sends map file names to *ASD* processes, and statistical information to the *FTM_Stats* process.

- map file name
- number of records
- TZ activity by ARTCC

Processing

The *Extract* module retrieves the flight data from the database every map cycle interval for distribution to map clients (ASD programs). This module sends the following statistics to the statistical display program (*FTM_Stats*) for a display on an Apollo node: map file output time, size data, and flight status data (i.e., numbers of active, ghostable, expired, and pending flights).

During each map cycle interval, the *FTM* estimates the current position and heading of all active aircraft in the database to be placed into the map file. First, the distance traveled since the calculation is determined by the last reported speed, the time of the calculation, and the current time. The current position is calculated by moving the flight toward its next waypoint from its last calculated position by the distance travelled. The next waypoint must be in the flight's waypoint list. The waypoint to be ghosted toward is validated by comparing the distance travelled to the distance to the waypoint. If the waypoint is farther than the distance travelled, it is used; otherwise, the next waypoint in the list is checked. This continues until a waypoint is validated or the list has been traversed. If no waypoints are validated, the flight is ghosted toward its destination airport.

Routines. The *Extract* module is composed of the following independent routines:

- *Build_Ascii_Altitude*, given a numeric altitude and an altitude type, provides an ASCII altitude and assigns it to the appropriate field in the database: **filed_ascii_altitude** if the boolean parameter **filed** is true, **ascii_altitude** otherwise.
- *Calculate Distance* is a function that provides the distance between **x** and **y** values using the *Great_Circle* module.
- *Calculate Location* fills in information for the *Flight_Table_Retrieve* routine. It attempts to calculate the current location for every active flight in the database.
- *Flight_Table_Retrieve* handles the extraction of data into the variable **map_record**. It sets flags for ARTCCs that have not sent any TZ messages since the last map file. When each entry has been processed, the **map_record** is written to a disk file by the routine *Write_to_Map_File*. When all entries have been processed, this routine closes the two files that make up a map file. The *Send_To_Clients* routine (in the *Misc* module) is called to send the map file names to all **maps** clients. A summary message is also sent by the *Stats_Send_Data* (also in the *Misc* module) to all **stats** clients.
- *Heading* determines an aircraft's heading using source and destination coordinates. It is set up for Albers projection but can be used for lat/lon by reversing the x-coordinates.
- *Map_Heading* is a function which returns a character based on the provided heading.
- *Write_To_Map_File* is given the name of a map stream. It checks if the stream value is 0 (stdout); if so, the *Read_Adapt_File* routine is called, and **map** and **rte** files are created with ios_$create, appending a date/time stamp to the filenames. Otherwise, it can be assumed that the filenames and streams are intact.

  If the global variable **bad_map** is true, then the routine is aborted. Next the global variable **map_route_output_rec** is written to the **map1_file_stream** (**rte** file) and **map_output_rec** is written to the given stream ID (**map** file) using the ios_$put call. **Map_number_entries** is incremented.

Error Conditions and Handling

System call failures cause error messages to be written to the trace log. Specific error messages are listed in Table 18-21.

18.1.4 The Flight_Handler Module

Purpose

This routine manages the flight table database by performing the following functions: add, delete, initialize, purge, rebuild, validate, and find. Refer to section 18.4 for more detailed information on the flight table data structure.

Input

When performing add, delete, or find actions, the *Flight_Handler* module receives an aircraft identifier (**acid**) for a flight; otherwise, it receives no input.

Output

The flight database is modified according to the action taken by the called procedures.

Processing

Figure 18-6 provides a diagram to help explain the organization for the *FTM* flight database. The flight table is stored as an array of records. Upon flight table initialization, the hash table is cleared, and each record is linked together in the allocated free space of the table. The hash table is an array of pointers, each of which points to a linked list of elements.

When a flight is to be located, the aircraft identifier of the flight is hashed, and the appropriate address in the hash table is traversed. Each entry that is traversed is a subscript of an element in the flight table (i.e., **flight_table_entry_t**). If the flight is not found, the entry is taken from the free space in the array and put at the end of the linked list for that hashed address.

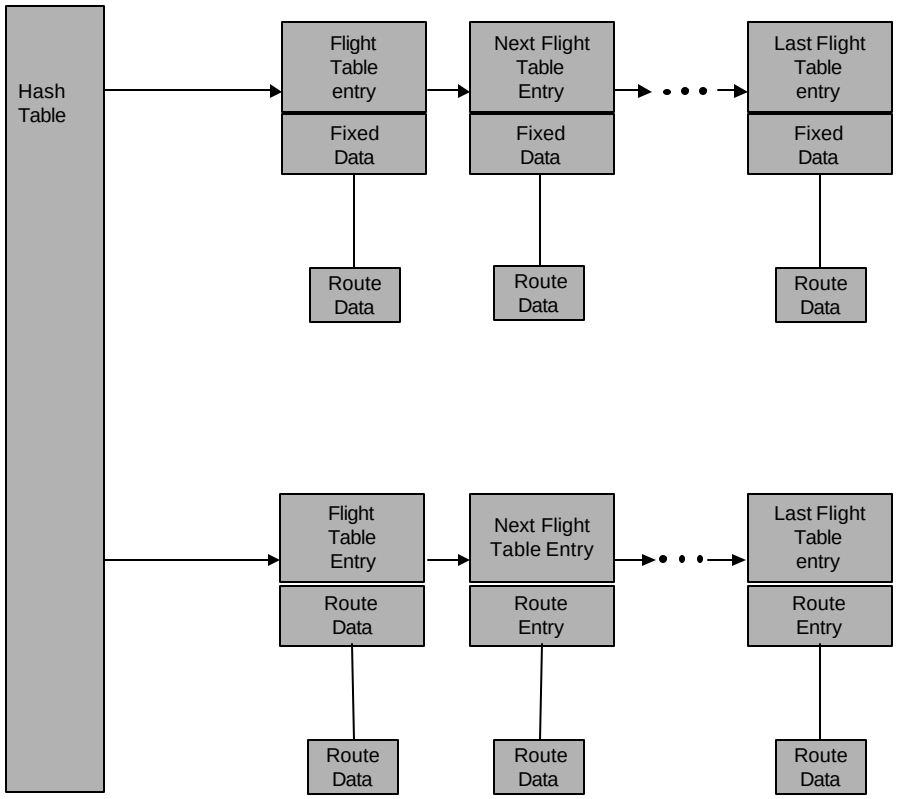

Figure 18-6. FTM Database Structure

Routines. The *Flight_Handler* module is composed of the following independent routines:

- *Activate_Flight* finds an available slot in the *Flight_Active* table by hashing the **acid**. If there are no entries to this linked list, space is allocated by the *Active_Storage_Allocate* routine. Otherwise, the linked list is traversed. If a match is found, that entry is updated else space is allocated by the *Active_Storage_Allocate* routine.

- *Active_Storage_Allocate* is a function which returns the index of the next available slot or a **-1** if the table is full. The search begins by examining the slot following **Flight_Active_Last_Slot**. If the bitmap has a zero for that location then that slot is returned and **Flight_Active_Last_Slot** and the bitmap are adjusted. If not, the next location is examined.

- *Create_Flight_Entry* calls *Flight_Table_Get_Next_One* to allocate storage. The **acid** is hashed. If there are no entries at this location, then it is added to the list header and the entry is set up. If there are entries at the hash location, each is examined to see if it has expired. If it has, then it is deleted.

  When the end of the linked list is found, the pointer to the end is set to the allocated space, and the entry is set up. The entry is set up by moving the **acid** value into the records **Flight_Array1** and **Flight_Array2**, along with creation, update, and deletion times; all other elements are set to their default values.

- *Deactivate_Flight* is given the **acid** to identify the flight. It hashes into the active table and walks across the linked list until it finds the entry. Once found, it uses the previous linked list address to link around the entry and clears the **in use** flag in the bitmap.

- *Delete_Active_Flight* is given the index, pointer, and previous entry for the element to be deleted. It links around the entry and clears the flag in the bitmap.

- *Delete_Expired_Table_Entries* walks through the entire flight table looking for entries whose deletion time has passed and deletes them. Any entries with an active flag of **X** or **2** are also deleted. For every 500 hash entries processed, it calls the *Check_Other_Activities* routine. *Delete_Expired_Table_Entries* is called every three map cycle intervals, and in response to the **PURGE** command. This routine takes a long time to run and causes a significant amount of page faulting.

- *Find_Flight_Acid* finds a flight in the active list by matching the 7-character aircraft identifier (**acid**) to match raw messages from the local site (DZs and TZs). It hashes the **acid** and traverses the linked list.

  The departure time must be within 45 minutes of that in the database. If a match is being searched with a DZ, then the airports are compared between the message and the database. If a match is found, the index is returned with the variable **successful** which is set to **zero**. If the search is unsuccessful, the variable is set to **one**.

- *Find_Flight_Acid_Oceanic* finds an international flight in the active list by matching the 7-character aircraft identifier (**acid**). It hashes the **acid** and traverses the linked list. The entry matching closest against the departure time, if one is found, is used. If a match is found, the index is returned with the variable **successful** which is set to **zero**. If the search is unsuccessful, the variables is set to **one**.

- *Find_Flight_Acid_Time* finds a flight in the active list by matching the 7-character aircraft identifier (**acid**). It hashes the **acid** and traverses the linked list. The list is searched for an entry where the input **wanted_time** falls between the departure time and the arrival time (using the routines *time_used_get_depart* and *time_used get_arrival*). If a match is found, the index is returned with the variable **successful** which is set to **zero**. If the search is unsuccessful, the variable is set to **one**.

- *Find_Flight_On_Index* takes a **flight_index** and a 7-character aircraft identifier (**acid**). It hashes the **acid** and traverses the linked list. If an entry is found with the **flight_index**, the index to the entry is returned. Otherwise, a nil entry is returned signifying that no match was found.

- *Find_Flight_On_Index_All* performs the same function as *Find_Flight_On_Index*, but also searches deactivated flights.

- *Flight_Table_Clear_Memory* clears and recreates the flight database. It creates and writes the **/ftm_check/shared_region_header** file, zeroes out the flight table's **in use** bitmap, zeroes out the route table's **in use** bitmap, clears the flight hash table, zeroes the active table's **in use** bitmap, clears the active hash table and calls *Airport_Header_Create_Table*.

- *Flight_Table_Extract_Field10* uses the provided flight pointer to determine the starting address of the appropriate route block. When this address is located, the necessary number of bytes are moved out. If another route block is needed, it is obtained. This process is repeated until all the data is retrieved.

- *Flight_Table_Extract_Fixes* uses the provided flight pointer to determine the starting address of the appropriate route block. When this address is located, the necessary number of bytes is moved out. If another route block is needed it is obtained. This process is repeated until all the data is retrieved.

- *Flight_Table_Extract_Sectors* uses the provided flight pointer to determine the starting address of the appropriate route block. When this address is located, the necessary number of bytes are moved out. If another route block is needed, it is obtained. This process is repeated until all the data is retrieved.

- *Flight_Table_Extract_Waypoints* uses the provided flight pointer to determine the starting address of the appropriate route block. When this address is located, the necessary number of bytes are moved out. If another route block is needed, it is obtained. This process is repeated until all the data is retrieved.

- *Flight_Table_Get_Next_One* is a function used to return the next available slot in the flight table. It starts with **Flight_Array_Last_Slot** and searches the **in use** bitmap for an unused position. When found, it updates **Flight_Array_Last_Slot** and marks the spot in the bitmap. However, if no available slot is found, then the table is full. *Send_Stats_Display* is called, and a **zero** entry is returned for the location.

- *Flight_Table_Release_Storage* is given a table address. First, it validates the address. If it is an active flight, it calls *Deactivate_Flight*. Then, it calls *Flight_Table_Rte_Release* and clears the **acid**, active flag, and next pointer in the flight table.

- *Flight_Table_Rte_Insert* adds route information to **ftm_checkpoint_b**. First it calculates the number of pages required (248 bytes each). If too many (i.e., 9), an error is returned. If the number of pages is the same as it was before, the old data is overwritten. If not the same, the bitmap is searched to find enough contiguous space. When found, the bitmap is marked, and the data is moved in.

- *Flight_Table_Rte_Release* is given a flight pointer. It clears out the control information, data, and bitmap for each page owned by that flight.

- *Flight_Table_Rte_Retrieve* moves out all bytes of route data pertaining to a specified flight.

- *Get_Flight_List* is given a 7-character aircraft identifier (**acid**), which is assigned to **flight_list_acid**. **Flights_in_list** is assigned **1** and **flight_list[1]** is set to **-1** to indicate no active flight. It hashes the **acid** to the **active_table** and traverses the linked list.

  With each entry, the **flights_in_list** global variable is incremented. If it finds an active entry (by checking for an active flag of **A** or **E),** it puts the active entry in the first position of the linked list. Next, the **flight_table** is hashed and the linked list traversed. Along the way, the **deletion_time** of the flight is checked to see if it is time to remove the entry from the database.

- *Get_Flight_List_All* is given a 7-character aircraft identifier (**acid**), which is assigned to **flight_list_acid**. **Flights_in_list** is assigned **1** and **flight_list[1]** is set to **-1** to indicate no active flight. The acid is hashed to the **active_table** and the linked list is traversed.

  With each entry, the **flights_in_list** global variable is incremented. If an active entry is found by checking for an active flag of **A** or **E,** the routine puts it in the first position of the linked list. Next, the **flight_table** is hashed and the linked list traversed. Along the way, the **deletion_time** of the flight is checked to see if it is time to remove the entry from the database. The difference between this routine and *Get_Flight_List* is that cancelled flights are also included in this flight list.

- *Hash_ACID_Active* is a function which hashes an **acid** to obtain an index into the active table.

- *Time_Used_Get_Arrival* sends back a code to indicate which of the five possible times is being used for the arrival airport. The codes are

  0 = not specified
  1 = actual
  2 = ttm/estimated
  3 = controlled
  4 = proposed
  5 = scheduled

- *Time_Used_Get_Depart* sends back a code to indicate which of the five possible times is being used for the departure airport. The codes are

  0 = not specified
  1 = actual
  2 = ttm/estimated
  3 = controlled
  4 = proposed
  5 = scheduled

- T*ime_Used_Set_Arrival* sets the time being used for the arrival airport with one of the following codes:

  0 = not specified
  1 = actual

2 = ttm/estimated
3 = controlled
4 = proposed
5 = scheduled

- *Time_Used_Set_Depart* sets the time being used for the departure airport with one of the following codes:

  0 = not specified
  1 = actual
  2 = ttm/estimated
  3 = controlled
  4 = proposed
  5 = scheduled

Error Conditions and Handling

The following error condition occurs in one routine in this module, if there is no room left in the flight table:

Create_Flight_Entry **Pfm_$Error_Trap**

A traceback is saved and written to the trace log. Specific error messages are listed in Table 18-21.

18.1.5 The Great_Circle Module

Purpose

This module calculates the polar spherical triangle for the great circle route between two longitude/latitude points: the origin and destination of a flight.

Input

Great_Circle input consists of latitude and longitude values for the two points.

- lat1 – latitude
- lon1 – longitude
- lat2 – latitude
- lon2 – longitude

Output

Great_Circle output consists of the distance (in nautical miles) and aircraft heading. When the calculated distance is zero, the returned heading is also zero.

- dist – distance
- hdg – heading

Processing

The *Great_Circle* module consists of one function, which computes the distance and heading for a flight. It is used if no new position information has been received after a specified amount of time for an international flight. It is the only routine written in Fortran.

Error Conditions and Handling

None.

18.1.6 The Interface Module

Purpose

This module provides the interface to the ETMS message switching system. This module processes incoming messages, sends outgoing messages, manages the client table, generates statistics and other reports, and registers for flight data.

Input

The *Interface* module accepts the following global variables:

- sw_buffer
- sw_size
- address_list
- address
- cnt

The following global variables are outputs of the *Interface* module:

- sw_buffer
- sw_size
- address_list
- address_cnt
- blocks_from_sw
- blocks_to_swbytes_to_sw
- current_time

Processing

Routines. The *Interface* module is composed of the following routines:

- *Add_A_Client* moves the global variable **sw_buffer** into a local buffer of type **net$_user_reg_with_provider_t**, a client registration structure. If the *service_count* field of the buffer is 0, then *Delete_A_Client* is called. If the **service_ count** is greater than **net$_max_services_needed**, an error message is sent to the requesting client, and the routine is aborted.

  The **services** field of the buffer is checked to determine the type of services the prospective client is requesting. The **client_table** is then traversed by the **address** field, checking if the prospective client already exists. If so, the new information overwrites the existing information for the client. If the client is not found to be in the table, it is added, and the appropriate global counters for each type of service are incremented.

  The *Put_Data* routine is called to notify the new client that it has been accepted. The global variables **client_count**, **stats_count**, **rpt_count**, and **raw_count** are incremented appropriately.

- *Air_Req* is given a **report_buffer** containing one or more airport identifiers. For each one, the *Find_Airport* routine is called to validate the airport and return its coordinates. The **report_buffer** is used as an output buffer where the airport

information is stored in a readable format, listing the coordinates (whether or not the airport is international) and the Alber's coordinates.

- *Check_Flight_Data_List* removes flight entries from the **flight_data_list** that are past their removal time.

- *Check_Sw_Mbox* processes mailbox input from other ETMS processes and forwards them to the appropriate handling routines. Messages are obtained using the *net$_get_message* call, which is repeated until no messages are found. Each message has a message type allowing the routine to delegate the message. Message types that are recognized are described below. The global variables **blocks_from_sw** and **bytes_from_sw** are incremented.
  - Transaction type messages are passed to the *Parse* module for parsing.
  - Client registration requests are passed to the *Add_A_Client* routine.
  - Responses to *FDBD* registration for data are handled by the *Process_Good_Reg* and *Process_Bad_Reg* routines.
  - Responses to *FTM* registration requests (when running as *FTMB*) is handled by the *Reg_With_FTM* routine (in the *Misc* module).
  - Requests for statistics are passed along to the respective statistic compiling routines, *Stats_Level_0* through *Stats_Level_9*.
  - Recovery protocol messages are handled by the *Recovery_Parse_Messages* routine (in the *Recovery* module).
  - User requests for specific flight data are handled by the *Pending_Queue_Push* routine (in the *Misc* module). See Table 18-2 for a description of all user requests. These commands are also known as f commands because they may be issued with the *Net.mail* f <command_name>, as well as from the *ASD* process via list requests. The chart also indicates which of the commands are sent to the *FTM_Coprocess* for processing.

- *Close_Clients* sends a **net$_msg_reg_closed** message to all clients and clears the **client table**.

- *Delete_A_Client* searches for the provided client in the **client_table** and, if found, removes it. The global variables **client_count**, **raw_count**, **stats_count**, and **rpt_count** are decremented appropriately.

Table 18–2. FTM Commands

Command	Description	Send to Coproc
ADRFILLIT	Expand ASD/ADR report with flight data	Yes
AIR	Display provided airport characteristics	
AIRPHEADER	Internal debugging command for airport header tables	Yes
ARR	Display arrivals from provided airport	Yes
ARRIVALS	Generate a file of arrivals at provided airport	Yes
ARRT	Display arrivals from provided airport ordered by time	Yes
BOTH	Generate a file of arrivals/departures at provided airport	Yes
DEP	Display departures from provided airport	Yes
DEPARTURES	Generate a file of departures at provided airport	Yes
DEPT	Display departures from provided airport ordered by time	Yes
DUMPACTIVE	Internal debugging command for active list	Yes
FLIGHTS	List all flights for a provided airline	Yes
HELP	Describe all FTM commands	
LIF	List all legs of a flight on ASCII	Yes
LIFP	List all legs of provided flights in binary	
PACE	Display pacing airport statistics	
PURGE	Remove all expired flights from database	
SHO	Show data for up to five flights	
SITE	Provide FDB site connections	No
TDBFILLIT	Expand TDB report with flight data	Yes
TIME	Return the time	
VAL	Same as PURGE	
VALIDATE	Internal debugging command for validating tables	Yes
0,1,2,3,4,5,6,7	Stats_Level_0 through Stats_Level_7	

- *Display_Flight* is given a flight pointer (**count**) and returns a buffer and **buffer_size**, containing the flight's information in a readable format. All useful database fields are included in the output.
- *Display_Lif_Flight* is given a flight pointer (**count**) and returns a buffer and **buffer_size**, containing summarized information about all legs of a flight,

including the arrival/departure airports and times and the routes. This routine is called in response to the F LIF *Net.mail* command.

- *Ftms_Process* handles user requests when an *FTM_Coprocess* is not able, calling the appropriate report generating routine for the command entered.
- *Help_Req* supplies an output buffer containing all F commands and their descriptions.
- *No_Data_Avail* is given an index in the **flight_data_list**. It calls *Put_Data* to send a message to the *FTM_Coprocess*, using **flight_data_list[index].coproc_addr**, indicating a queued request for flight data that was not fulfilled.
- *Open_Sw_Mbox* opens a connection to the *Nodeswitch* process with the net$_open call, after checking whether to register as *FTM* or *FTMB*. This allows *FTM* to communicate with other connected processes in the ETMS. If the net$_open attempt fails, a second attempt is made. Otherwise, the **registered_prov** global variable is set to **TRUE**.
- *Process_Bad_Reg* handles the receipt of a bad registration message from the *FTM*'s data source. If running as *FTMB*, then the *Register_To_Provider* routine is called to retry registration to the master *FTM*. If running as an FTM, then the *Register_To_Provider* routine is called to retry registration to the *FDBD*, the **reg_failed** global variable is incremented, and the **conn_start_time** global is reset to the **current_time**.
- *Process_Good_Reg* handles the receipt of a registration acceptance from the *FTM*'s data source. If running as *FTMB*, the **registered_serv** global is set to **TRUE**, the **registration_outstanding** global is set to **FALSE**, and the **no_data_time** global is set to the **current_time** + **no_data_timeout**.

  If running as *FTM*, the source site of the message is compared to the **current_site_id** for validation. If the message is from a different site, the registration_acceptances ignored global is incremented and the *Register_To_Provider* routine is called with a **0 service_count** to close any registration with the unwanted provider. If the sites match, then the class is validated to ensure the message came from an *FDBD*. The **reg_success** global is incremented, the **providers_addr** global is set to the source address of the message, the **current_site** is set via the net$_inq_get_site_ascii call, the **registered_serv** global is set to **TRUE**, and the **no_data_time** is set to the **current_time** + **no_data_timeout**.
- *Process_Incomplete_Request* handles user requests attempted to be handled by *FTM_Coprocess*, which was not able to complete due to holes in the database. This routine increments the global **rept_rcvd**, checks that the size of the request buffer is valid, and calls the *Parse* module routine *Fill_In_Flights* to request the missing data from the *FDBD*.

- *Process_Reconfigure* handles reconfigure requests in one of two ways: reading a provided configuration file or resetting the primary and secondary FDBD registration sites to the provided sites. To determine the type of reconfigure, it checks the first character. If it is not a **$**, then it considers the input to be a filename; otherwise, it reads the site(s) provided and assigns them to **primary_site** and **secondary_site** respectively.

  If no **secondary_site** is provided, it is assigned the same value as **primary_site**. For filename reconfigurations, the *Misc* module routine *Read_Adapt_File* is called. If the reconfigure causes the *FTM* to be changed to an *FTMB*, or vice versa, then the net$_close call is used to close the registration to the *Nodeswitch* process, and net$_open is called to re-register as the new class.

- *Process_Returned_Message* handles messages that were attempted to be sent from *FTM*, but did not reach their destination, and returned to the *FTM*. If the message was intended for a client, *Delete_A_Client* is called to remove the client from the **client_table**.

- *Put_Data* sends a message to an **address_list**, using the provided message code (**mcode**). It presumes that the **address_list** and **address_count** were set before it was called, and uses the net$_send_message_addr_list call. It also checks if the **sw_handle** is nil, and, if so, calls *Open_Sw_Mbox*.

- *Register_To_Provider* sends a registration message (with a count of **0** to close registration and **1** to open registration) to the provided address. If running as *FTMB*, then the net$_inq_class_on_site call is made to find an *FTM* on the local site with which to register, unless the count is **0**, in which case the net$_connect_to_service_provider call is made. If running as *FTM*, the **providers_addr** is assigned the provided address and the net$_connect_to_service_provider call is used to send the request. The **service_time** is set to the **current_time** + **retry_time**. A check is made to ensure that the destination address site is not **net_nil**.

- *Sho_Req* handles requests for the **F SHO** command, which allows up to five flight acids to be requested for display. It calls the *Display_Flight* routine to write each flight's display into a buffer which is sent to the requestor via the *Put_Data* routine.

- *Site_Req* responds to the **F SITE** command by creating a buffer containing the FDB site connections and calling *Put_ Data* to return this information to the requestor.

- *Stats_Level_0* sends a buffer stating that "This FTM command is not yet supported..." to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**. When a stats level 0 command is issued to *FTM*, the *Stats_Level_1* routine is called.

NOTE: This procedure is not called, because statistics levels 0 and 1 are currently required to be equivalent.

- *Stats_Level_1* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_2* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_3* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_4* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_5* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_6* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_7* writes statistics about *FTM* to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_8* writes statistics about the five most recent reconfigured requests to a buffer to be sent to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stats_Level_9* sends a buffer stating that "This FTM command is not yet supported..." to the requestor via the *Put_Data* routine. The requestor's address is stored in **address_list[1]** from **sw_header.source_address**.
- *Stop_Recovery_On_Previous_Site* is given a **site_id**. It increments the **recoveries_aborted** field of the global **recovery_record** and sets the **current_state** of the record to **no_recovery_in_progress**. The provided **site_id** is incorporated into **address_list[1]**, along with the *FDBR* class name, and the *Put_Data* routine is called to send an ftm$_t_recovery_stop message to the *FDBR*, which in turn will stop sending recovery messages to the *FTM*.
- *Switch_Sites* switches the **providers_addr** (contained in the *FDBD*) from the **current_site** to the other site; i.e., if currently connected to the **primary_site**, this routine switches to the **secondary_site**, and vice versa. If the primary and secondary sites are the same, then no site switch is made. The site switch is performed by calling *Register_To_Provider*.

Error Conditions and Handling

Error messages, such as network addressing calls and system calls that are unsuccessful, are written to the trace log. Specific error messages are listed in Table 18-21.

18.1.7 The Misc Module

Purpose

This routine performs various utility functions required by the *FTM*.

Input

The *Misc* module receives data to be logged or distributed.

Output

The *Misc* module produces the following types of output:

- Statistics, sent to the *FTM_Stats* program
- **Map**, **rte**, and **raw** data
- Data that is sent to disk

Processing Routines.

The *Misc* module is composed of the following routines:

- *Albers* converts latitude and longitude into an Alber's projection.
- *Arcsin* is a function which computes the arcsine of a number.
- *Check_Other_Activities* is called to perform real-time operations. It calls the *Timer_Process* and *Check_Sw_Mbox* routines (in the *Interface* module). Then, if not making a map or a report, it calls *Pending_Queue_Pop*. Then it calls the *Delete_Expired_Table_Entries* routine (in the *Flight_Handler* module).
- *Check_Truncate_File* checks the size of the trace log and truncates it if it is too large.
- *Compare_Strings* is a function that determines if two strings are equal.
- *Display_Error* writes the text associated with a status code along with a supplied text to the trace log. This data is also sent to **stats** clients, and, if **send_log** is set, to a *Logger* process.
- *Display_Net_Error* writes the text associated with a toolkit status code along with a supplied text to the trace log. This data is also sent to **stats** clients, and, if **send_log** is set, to a *Logger* process.
- *Get_New_Ecs* obtains new event counts when a node reopen occurs, using the *net$_get_ec_ptr* call.

- *Initialize* performs various setup functions. It calls *get_etms_path* to get the name of the path to prepend to all program related objects. *Set_Timer* is called, the trace log stream is created (moving the old trace filename to .bak), the *Recovery_Clear_Record* routine (in the *Recovery* module) is called, global variables are initialized, and the program argument is read - the adapt filename.

  The *Open_Sw_Mbox* routine (in the *Interface* module) is called, the data in **airstrip.dat** is read, and the *Misc* module routine *Read_Adapt_File* is called. Finally, it calls the *Ms* routine *Flight_Table_Create* to load the **Shared Region** and **Pacing_Clear_Table** to validate the airport header tables, and *Read_Key_File* to obtain NRP information.

- *Pending_Queue_Pop* takes the top entry off the pending queue and calls a routine to process it. All requests for data from *FTM* are placed on a pending queue. Periodically, this routine is called to process a request for data. If there is nothing on the queue, the **report_in_progress** flag is cleared and the routine returns. Otherwise, the contents of the current mailbox are saved, and the top entry of the queue is moved into these variables.

  The process type is examined. If the process type is to be handled by *FTM_Coprocess*, the routine *Send_To_Rpt_Process* is called. The *Interface* module routine *Ftms_Process* is called otherwise or upon failure to find an *FTM_Coprocess*. *Pending_Queue_Pop* disposes the **report_pending_queue** to free up the storage space allocated for the completed entry. The **report_in_progress** flag is cleared.

- *Pending_Queue_Push* examines requests and attempts to send them to the *FTM_Coprocess.* If the **report_in_progress** and **making_map** flags are both not set, *Pending_Queue_Pop* is called before the routine returns.

- *Process* is the main processing routine in *FTM*. It is initialized by a call to *Timer_Arm_It*. Then it enters an endless loop where it calls *Check_Other_Activities* and waits for any event counter to be exceeded. Also, data and connection timeouts are checked every time through the loop. The following paragraphs describe the algorithm used to request data from the *FDBD* and for switching sites.

  Registration Request Timeout

  - At initialization, FTM registers for services to the FDB on the primary site. If the *registration accept* message is not received within one minute, FTM sends a new registration request to the primary site. This one minute timeout/retry takes place until the *registration accept* message is received or until three total minutes have passed since the first registration attempt. If there is still no connection by the 3-minute mark, FTM switches to attempt registration on the secondary site. These attempts also allow one minute before re-registering, and three total minutes before switching sites again.

No Data Received Timeout

- Once the FTM is registered for services, the amount of time since the last received data message is monitored. If a minute passes without FTM receiving any data, FTM sends a registration attempt to the FDB on the primary site. If FTM had been registered for services on the secondary site, it sends a close registration to that site. At this point, the Registration Request Timeout logic in the above paragraph is used.

- FTM also tracks the number of times that a No Data Received Timeout occurs on the primary site. If this occurs three times, then the next No Data Received Timeout will cause FTM to switch to the secondary site. In other words, a No Data Received Timeout causes FTM to register to the primary site regardless of the current site, unless three consecutive registrations lead to No Data Received Timeouts, in which case the secondary site is used.

- Within the main processing loop, the timeouts are checked as follows:

```
if no data within 2 minutes then
                    if currently on secondary site then
                       switch sites to primary and register
                    else
                        increment no data count
                        if no data count = 3 then
                           switch sites to secondary and register
                        else
                           re-register to primary site
else if no registration in 3 minutes then
                    switch sites and register
else if no registration in 1 minute then
                    resend registration request
```

- *Read_Key_File* reads the remarks keywords file and stores this information for later use. The filename is **/etms5/shared/data/remarks_keywords**.

- *Reg_With_FTM* uses net$_connect_to_service_provider to register to another *FTM* as a backup *FTM* (*FTMB*). The **ftm_reg_address** is obtained from an inquiry for all *FTM*s on site.

- *Send_To_Clients* is given a client type (**ctype**) and a message type (**mtype**). It traverses the **client_table** for all clients matching the **ctype**, and it builds the **address_list** to which the *Interface* module routine *Put_Data* is called to send an **mtype** message.

- *Send_To_Rpt_Process* sends a user request to a **rept** client (*FTM_Coprocess*) for processing using *Put_Data* with message code ftm$_t_report_req. It is given a character code (**R** = resend the message, **Q** = queue) and outputs a boolean success (**0** = success, **1** = no **rept** client found).

- *Set_Timer* gets the current time (**timer**) and adds **time_correction** to obtain the value for the variable **current_time**. If midnight has been crossed, the variable **time_at_midnight** is reset. All time in *FTM* is based upon the **timer** and **current_time** variables. *Set_Timer* provides a common place to ensure that both are at the most current values.
- *Sincos* returns the sine and the cosine for the specified degrees.
- *Stats_Send_Data* is given a code and some text. It puts both items into a mailbox message and sends it to all channels with process type *Stats.* The value of code is as follows:

  01 – ARTCC statistics
  02 – pacing airport interval data
  03 – map statistics
  04 – parsing errors
  05 – miscellaneous statistical information
- *Timer_Arm_It* calculates the number of seconds until end of the next map cycle interval. The global variable **map_creation_time** is set. An event counter is set to go off ten seconds after this time to force a map file creation.
- *Timer_Process* examines the value of the event counter that went off. If it was the pacing airport 15-minute event, this routine calls *Pacing_Advance_Interval* and *Airport_Header_Advance_Interval*. But if it was the map time event, this routine calls *Flight_Table_Retrieve* and calls *Timer_Arm_It*. If it was any of the other event counters that went off, this routine returns. This routine also checks whether *FTM* has been waiting for recovery data for over an hour without receiving any. If so, the recovery is aborted and is restarted if less than 150 buffers were received in the previous recovery.
- *Trace_back* is called by the FTM *Main* module upon termination. It creates a file called **ftm_trace_log**. Then it invokes the **errlog_$traceback** command; the results are written to the file.
- *Validate_Class_And_Site* validates the **providers_addr** (FDBD) and the **src_ftm_addr** (FTM).

Error Conditions and Handling

The *Misc* routines write various error messages to the trace log in response to the network addressing system and system call failures. Specific error messages are listed in Table 18-21.

18.1.8 The Ms Module

Purpose

This module handles the flight database creation, mapping, and unmapping.

Input

Inputs to shared regions include **ftm_checkpoint_file_a** through **ftm_checkpoint_file_g**.

Output

Outputs to shared regions include **ftm_checkpoint_file_a** through **ftm_checkpoint_file_g**.

Processing

Routines. The *Ms* module is composed of the following routines

- *Flight_Table_Create* sets up the seven checkpoint files (**ftm_checkpoint_a to ftm_checkpoint_g**). First, this routine attempts to map these files. If there is an error, any files successfully mapped are unmapped, and the seven files are deleted along with the **shared_region_header** file, and all pointers to the shared region are set to zero. Then, the seven files (now empty) are mapped. If successful, *Flight_Table_Clear_Memory* in the *Flight Handler* module is called. If not successful, *FTM* calls the Aegis system call *Pfm_$signal* and exits.
- F*light_Table_Unmap* unmaps each of the seven checkpoint files from memory and updates the **shared_region_header** file.
- *Clear_it* is explained in the Error Conditions.

Error Conditions and Handling

Flight_Table_Create will display a message, unmap the checkpoint files and call the *Ms* module routine *Clear_it* if it gets an error trying to map one of the old checkpoint files or if it gets an error trying to open the **shared_region_header** file. If it gets an error trying to create new checkpoint files, it displays a message, calls the *Ms* module routine *Clear_it*, and then calls the Aegis system call *Pfm_$Signal*.

Flight_Table_Unmap will display an error if it encounters an error while unmapping any of the checkpoint files, or while opening the **shared_region_header** file, or while writing to the **shared_region_header** file.

18.1.9 The Pacing Module

Purpose

This routine maintains and processes statistics for the pacing airports, which are the 29 airports whose traffic sets the pace of all NAS air traffic.

Input

The *Pacing* module receives data to be written to the pacing tables.

Output

The *Pacing* module sends reports to a requesting program or user.

Processing

Pacing maintains tables on expected arrivals and departures at the pacing airports. It produces statistical reports showing the arrival and departure data for these airports. These reports are sent to the statistical display program (*FTM_Stats* process*)*. The pacing airport tables are initialized upon *FTM* initialization.

Routines. The *Pacing* module is composed of the following routine:

- *Find_Airport_In_Pacing* (see below)
- *Pacing_Add_Counter* checks the value of the pacing event counter (15 minutes). If 15 minutes have been exceeded, it calls *Pacing_Advance_Interval*. Then this routine calls the *Pacing* module routine *Find_Airport_In_Pacing*. If the event counter is not found, the routine returns. The correct time interval (bucket) is found by walking through the pacing table. Finally, two counters are incremented. One is for the pacing interval, and the other is for the pacing interval by aircraft category. There is also a set of counters for arrival and a set for departure. A boolean variable is passed in to tell this routine which set to update.
- *Pacing_Add_Future_Counter* updates counters for flights that have not occurred. First, it checks the value of the pacing event counter (15 minutes). If it has been exceeded, it calls *Pacing_Advance_Interva*l. Then this routine calls the *Pacing* module routine *Find_Airport_In_Pacing*. If the event counter is not found, it returns. The correct time bucket is found by walking through the pacing table. Finally two counters are incremented. One is for the future pacing interval, and the other is for the future pacing interval by aircraft category. There is also a set of counters for arrival and a set for departure. A boolean variable is passed in to tell this routine which set to update.
- *Pacing_Advance_Interval* resets the pacing event counter to go off in 22 minutes (15-minute interval plus 7-minute offset). Next, it opens **/traffic/ftm_pace_log** and writes the data from the interval that has just completed. A statistical summary is sent to all *Stats* channels. Finally, the intervals are adjusted to eliminate the oldest and to clear the latest.
- *Pacing_Clear_Table* fills the pacing table from the airport header table, and initializes all elements other than *name* to zeroes. Finally this routine sets the pacing event counter to execute in 15 minutes and calculates the starting time of each interval.
- *Pacing_Report* writes to the trace log a report of all the information contained in the pacing table. This routine is called to satisfy the **Pace** command.

Error Conditions and Handling

Only the *Pacing_Advance_Interval* routine contains error handling. It calls *Display_Error* if there is an error from any of the following calls used on the **ftm_pace_log** file:

- Ios_$Create

- Ios_$Inq_Byte_Pos
- Ios_$Seek

18.1.10 The Parse Module

Purpose

This routine processes and validates transaction messages from the *FDBD* process. Processed flight data is entered into the *FTM* database.

Input

The *Parse* module receives the following transaction data from the *FDBD* process:

- TZ_Data
- TTM_FTM
- critical
- cancel
- Block_Alt
- position
- time
- route

Output

The *Parse* module enters flight data into the *FTM* database. All messages received are sent to all **raw** clients.

Processing

The routines in this module parse the transaction messages. The information in these messages is used to update or create entries in the *FTM* database.

Routines. The *Parse* module is composed of the following routines:

- *Bad_Field* is given an error code, **int_value**, **num_msg**, and place. The global **parse_errors** is incremented. A case statment determines the error code, and a corresponding text message is encoded into a local buffer describing the error, using the other provided parameters.
- *Compute_Ete* is given a **flight_ptr** to the active flight table, an **arrival_ptr** and a **depart_ptr**. If **arrival_ptr = 0,** the *Flight_Handler* module *Time_Used_Get_Arrival* routine is called to determine which of the arrival times is being used. If **depart_ptr = 0,** the *Flight_Handler* module

Time_Used_Get_Departure routine is called to determine which of the departure times is being used.

If either of these times is still undetermined, the routine is aborted. Otherwise, the **arrival_time** and **depart_time** are set to the determined times, respectively. The estimated time enroute (ete) in minutes is determined by subtracting the **depart_time** from the **arrival_time** and dividing by 60. This value is assigned to the **ete** field in the database.

- *Convert_To_Hhmmss* takes a total seconds value and converts it into a text string, represented by **hh:mm:ss** (hours:minutes:seconds).

- *Create_Log_File* checks to see if the **orig_filter** is set. If so, *ios_$create* is used to create an orig file under the name **orig.MMddhhmmss**, by first using *cal_$decode_time* and *vfmt_$encode10*. The **log_reopen_time** is reset to be on the hour for the next orig file. Also, this routine resets all hourly buffer stats.

- *Data_Log* takes a packed transaction data buffer and size, checks if the **orig_filter** is on, and, if so, writes the buffer into the orig file.

- *Fill_In_Flights* is given a boolean queue, **num_flights**, and a **flight_list –** a record structure used to pass flight hole-fill requests to the *FDBR*. If queue is **FALSE**, the routine is aborted. (This is temporary until it is decided that automated hole fills may be enabled.) If a recovery is in progess, or has been requested, the routine is aborted. **Address_list[1]** is set to the *FDBR* address.

  If queue is **TRUE,** it means that the hole-fill request was sent from a list request which found a hole in the database, and so **flight_data.coproc_addr** is set to **sw_header.source_address** and **flight_data.seq_num** is set to **coproc_buffer.seq_num**. Otherwise **flight_data.coproc_addr** is set to **null_address** and **flight_data.seq_num** is set to **0**.

  Flight_data.id_flight is then set to **flight_list**, **flight_data.num** is set to **num_flights**, **flight_data.time** is set to **current_time**, and **num_flights_request** is set to **num_flights_request** + **num_flights**. The *Interface* module *Put_Data* routine is called to send the data request to the *FDBR* using the **ftm$_t_send_data** message type.

- *Find_Airport* attempts to return the latitude and longitude of a specified airport. First it checks to see if the airport code is prefixed with a **K**, which would mean that it is a Contiguous United States (CONUS) flight. If it is, it is removed. Then the airport table is searched. If the code is not found, **null** values are returned.

- *Get_Center_ID* is a function which translates a given **character_code** into its corresponding **center_id** using a case statement.

- *Increment_Source_Type_Stats* increments the number of messages received for the given **src_type**, which represents the source of a message (usually an ARTCC), in the **src_type_stats** array.

- *Insert_Route* calculates the size of the route field, saves the route counters, calls the *Flight_Handler* module routines *Flight_Table_Retrieve* and *Flight_Table_Insert*, and then restores route counters.

- *Parse_Block_Alt* parses a **block_alt** transaction message and then uses the *Flight_Handler* module routines to perform the following. It increments the global **block_alt_count**, calls *Increment_Source_Type_Stats* routine, validates the **acid**, and calls *Get_Flight_List* with the **acid**. If **flights_in_list** is > **1,** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **block_alt_match** is incremented. The database is updated with the message data.

- *Parse_Cancel* parses a cancel transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **cancel_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**. If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **cancel_match** is incremented. The database is updated with the message data.

- *Parse_Critical* parses a critical transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **critical_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **critical_match** is incremented. The database is updated with the message data.

- *Parse_Position* parses a position transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **position_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **position_match** is incremented. The database is updated with the message data.

- *Parse_Route* parses a route transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **route_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If still no match, then the orphan list is checked by calling *Find_Flight_On_Index* with the **orphan_index** of **-2**. (Orphans are created when FTM receives a raw message which it cannot match in the database.) If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **route_match** is incremented. The database is updated with the message data.

- *Parse_Time* parses a time transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **time_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, *Create_Flight_Entry* is called to create a new flight. If a match is found, then **time_match** is incremented. The database is updated with the message data.

- *Parse_TTM_FTM* parses a **ttm_ftm** transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **ttm_ftm_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **ttm_ftm_match** is incremented. The database is updated with the message data.

- *Parse_TZ_Data* parses a **tz_data** transaction message and then uses the *Flight_Handler* module routines to process the following. It increments the global **tz_data_count**, calls *Increment_Source_Type_Stats*, validates the **acid**, and calls *Get_Flight_List* with the **acid**.

  If **flights_in_list** is > **1** then *Find_Flight_On_Index* is called to try to match the message to an existing flight in the database. If still no match, then the orphan list is checked by calling *Find_Flight_On_Index* with the **orphan_index** of **-2**. (Orphans are created when FTM receives a raw message which it cannot match in the database.)

If no match is found, then an attempt is made to match the flight with the date bit (17th lsb) flipped (local procedure *Check_For_Date_Bit*). If still no match, then *Create_Flight_Entry* is called to create a new flight. If a match is found, then **tz_data_match** is incremented. The database is updated with the message data.

- *Process_Data* takes packed transaction messages from the *FDBD* and validates that the **sw_header.source_address.site_id** equals **current_site_id** and then uses the Recovery module routines to process. If the sending class is *FDBR* (indicating a recovery message), and the **site_id** does not equal the **current_site_id**, then *Stop_Recovery_On_Previous_Site* is called to abort the recovery from a different source site. The **msg_stats_total** record is updated, the *Data_Log* routine is called, and then a *while loop* is used to get each message, unpack it using *UnpackforFTM*, and call the approporiate parsing routine. If there was recovery data found, then the **recovery_record** is updated accordingly.
- *Process_Data_Unavailable* handles the situation where a hole fill was requested, but the data cannot be found. The **coproc_buffer** entry is cleared, and the *Interface* module routine *Put_Data* is called to notify the requestor that the data is unavailable, using the message type **ftm$_t_unavail_flts**.
- *Set_Times_Used* takes a **flight_ptr** and uses a hierarchy to determine the arrival and departure times to be used for the flight, calling the *Flight_Handler* module routines *Time_Used_Get_Depart* and *Time_Used_Get_Arrival*, and the *Parse* module *Compute_Ete* routine.The hierarchy is actual, controlled, estimated, proposed, and scheduled.

Error Conditions and Handling

Error conditions resulting from system call failures are written to a trace log.

18.1.11 The Raw Module

Purpose

The *Raw* module parses raw TZs and DZs and incorporates them into the *FTM* database.

Input

Raw TZ and DZ data buffers are passed to this module from the *Interface* module.

Output

The *FTM* database is modified according to the data received. Raw data files are created hourly when raw data is being received.

Processing Routines.

The *Raw* module is composed of the following routines:

- *Encode_Time_In_Seconds* receives text representing the date and time, and converts it into total seconds.
- *Parse_Field* receives a buffer **txt** with its size **text_size**, a pointer **ptr** to the text, and a **field_id**. *Parse_Field* returns a word, **word_size** and a boolean error. The **field_id** is determined in a case statement, and the corresponding field is parsed. The result is passed back in word and **word_size**, and error is returned as **TRUE** if one of the parsing rules failed.
- *Process_Raw_DZ* receives a raw DZ and a timestamp, calls the *Parse* module *Parse_Field* routine for each word separated by spaces and if no errors are found, it calls the following *Flight_Handler* module routines. The *Increment_Source_Type_Stats* routine attempts to match the data into an existing flight by calling the *Get_Flight_List* and *Find_Flight_Acid* routines. If a match is found, the flight in the database is updated with the new information. Otherwise, an orphan flight is created with a **flight_index** of **-2**.
- *Process_Raw_TZ* receives a raw TZ and a timestamp, calls *Parse_Field* for each word separated by spaces, and, if no errors are found, calls *Increment_Source_Type_Stats*, and attempts to match the data into an existing flight by calling *Get_Flight_List* and *Find_Flight_Acid*. If a match is found, the flight in the database is updated with the new information. Otherwise, an orphan flight is created with a **flight_index** of **-2**.
- *Process_Raw_Message* receives a buffer of one or more raw messages. The buffer is moved into a local buffer (**bufferl**), and the first 8 bytes of the buffer are validated as the password. If the password is not valid, the routine is aborted. Otherwise, the global **raw_buff_rcvd** is incremented.

  The buffer is repeatedly read one message at a time, using the line feed character as the delimiter. Each buffer has its 4-byte timestamp stripped and stored into a local variable timestamp. The 5th byte is taken as the ARTCC identifier. Then the two bytes representing the message type are checked.

  If TZ, then *Parse_Raw_TZ* is called with the message and timestamp (the **center_char** is global to the Raw module). If DZ, then *Parse_Raw_DZ* is called with the message and timestamp. Otherwise, the global **invalid_raw_count** is incremented. This *repeat-until loop* continues until there are no more messages found. All messages are also written to the hourly raw data archive file.

Error Conditions and Handling

In response to system call failures, error messages are written to the trace log.

18.1.12 The Recovery Module

Purpose

This module maintains the recovery state, updates recovery history information, sends recovery requests to the *FDBR* process, and handles a recovery message protocol with the *FDBR.*

Input

The global record **recovery_record**. Its structure is described in Table 18-3.

Table 18–3. recovery_record Data Structure

recovery_history_t				
Library Name: etms_lib		**Purpose:** Contain recovery information		
Element Name: ftm_constants.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
number_recoveries	Number of recoveries completed since FTM started.			integer32
number_recovery_ failures	Number of recovery attempts failed.			integer32
number_recovery_ attempts	Number of recovery attempts.			integer32

Table 18–3. recovery_record Data Structure (continued)

recovery_history_t				
Library Name: etms_lib		**Purpose:** Contain recovery information		
Element Name: ftm_constants.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
recoveries _aborted	Number of recoveries aborted.			integer32
total_recovery_time	Total time FTM spent in recovery mode.			integer32
total_buffers_rcvd	Total recovery buffers received.			integer32
total_msgs_rcvd	Total recovery messages received.			integer32
total_bytes_rcvd	Total recovery bytes received.			integer32
current_buffers_rcvd	Number of buffers received during the current recovery.			integer32
current_msgs_rcvd	Number of messages received during the current recovery.			integer32
current_bytes_rcvd	Number of bytes received during the current recovery.			integer32
previous _buffers_rcvd	Number of buffers received in previous completed recovery			integer32
previous _msgs_rcvd	Number of messages received in previous completed recovery			integer32
previous _bytes_rcvd	Number of bytes received in previous completed recovery			integer32
previous _start	Start time of the previous completed recovery.			integer32
previous _stop	Stop time of the previous completed recovery.			integer32
previous _time_range	Interval of time recovery requested for previous recovery.			array[1.2] of integer32
previous _state	Recovery state in previous recovery attempt.	**		recovery_state_t
current_state	Recovery state in current recovery attempt.	**		recovery_state_t
current_start_time	Start time of current recovery session.			integer32
current_time_range	Time interval of current recovery request.			array[1.2] of integer32
last_buffer_rcvd	Time of last recovery buffer received.			integer32
current_recovery_ reason	Reason for current recovery attempt.	**	0...13	integer16
previous _recovery_ reason	Reason for previous recovery attempt.	**	0...13	integer16

```
**recovery_state_t = no_recovery_in_progress,
                     recovery_should_be_started,
                     recovery_req_sent,
                     recovery_accepted_by_fdb
                     recovery_queued_by_fdb,
                     recovery_in_progress,
                     las_recovery_aborted
                     recovery_timed_out_restarting,
                     recovery_timed_out_not_restarting
```

recovery resons:	0	none
	1	database not found or error mapping table a
	2	error mapping table b
	3	error mapping table c
	4	error mapping table d
	5	error mapping table e
	6	error mapping table f
	7	error mapping table g
	8	start count <> stop count
	9	error reading shared region header
	10	shared region over an hour old
	11	last FTM shutdown over 12 hours old
	12	last FTM shutdown over 15 minutes old
	13	no data received for at least 15 minutes

Output

The **recovery_record** is modified as needed by the *Recovery* routines.

Processing

The following decribes the different types of recoveries:

Full Recovery (Database Wipeout). *FTM* requests a full recovery of NAS data when it detects that the database is invalid. In this case, *FTM* clears its current database and populates a new one with the incoming recovery data received from *FDBR*, as well as the normal data from *FDBD*. The time frame of this recovery will be [BEGINNING, Current Time]. The *FTM* determines that its database is invalid during *FTM* startup, while reading the shared region under the following conditions:

- If the database is over 12 hours old (retention period)
- An error occurs while mapping in one of the files making up the shared region
- The shared region header file indicates that the number of *FTM* starts does not match the number of *FTM* stops since the region was created
- The database files do not exist.

Finite Recovery. If a 15-minute period goes by without *FTM* receiving data from the *FDBD*, the *FTM* requests a finite recovery with a time frame of [Last Time Data Received, Current Time]. This request is made upon receiving a registration acceptance message from *FDBD.* A finite recovery request is also made upon FTM startup if it is determined that it has been over 15 minutes since the previous *FTM* shut down. In this case, the time frame is [Time of Previous *FTM* Shut Down, Current Time]. Another scenario causing a finite recovery is if *FTM* starts and does not receive data for over 15 minutes, a recovery will be initiated upon receipt of the next *FDBD* registration acceptance message. This recovery time frame is [*FTM* Start Time, Current Time].

Automated Interim Recovery (currently disabled). *FTM* processing of TZ, **block_alt**, time (AZ, DZ, EDCT, 5 setback), and position (TO, TA) type messages checks a **route_flag** sent from *FDBD* as part of the message, which indicates whether route data is available on the flight. If this flag is set and *FTM* does not have the route data, an interim recovery request (hole-fill) will be sent to the *FDBR.* Since FTM can pack up to 100 of these requests in one buffer, it sends one package of requests per message buffer received (as long as there is at least one flight in need of route data). If more than 100 flights are found needing route data in a single message buffer, *FTM* sends extra request buffers accordingly. The resulting TTM messages from *FDBR* provide *FTM* with the missing route data.

List Request Interim Recovery. A list request interim recovery occurs while *FTM* or *FTM_Coprocessor* is processing a list request and determines that flight data is missing. The *FTM* then sends *FDBR* a request for all data on the flight in question. Once the resulting TTM_FTM message is received containing the flight's information, the list request is completed.

Other *FTM* recovery notes: After one hour of successive unsuccessful recovery attempts, it sends an "ftm$_recovery_stop" message to the *FDBR*, which then aborts the recovery. If *FTM* switches registration sites during a recovery, it sends an "ftm$_recovery_stop" message to the *FDBR*, which then aborts the recovery. During *FTM* startup, if the database is over one hour old, the active table is cleared.

Routines. The *Recovery* module is composed of the following routines. Unless otherwise noted, the variables referenced below are fields of the global record **recovery_record**:

- *Recovery_Clear_Record* initializes the fields of the recovery record, specifically that the **current_state** is **no_recovery_in_progress** and that no recoveries have been attempted, completed, etc.
- *Recovery_Encode_Report* receives a buffer and **buf_ptr** and writes the current recovery information into the buffer starting at **buf_ptr**. All pertinent information is provided, i.e., previous recovery information is included only if there was a previous recovery completed. This routine is called from the *Interface* module routine *Stats_Level_4*.

- *Recovery_Needed* receives a start time and a recovery reason. It sets the **current_recovery_reason** in the **recovery_record** to the given reason. An entry is made into the trace log stating the reason. If the **current_state** is **recovery_should_be_started** then **current_time_range[1]** is updated to the given start time (unless it is less than the given start time). The **recovery_state** is set to **recovery_should_be_started**. If the **recovery_state** is not **no_recovery_in_progress** then the routine is aborted. The **current_start_time** is set to the current time.

- *Recovery_Parse_Messages* checks the global **sw_header.message_type** and determines if it matches one of the following recovery message types:
  - For ftm$_t_full_recovery, a message is written into the trace log stating that ftm should never receive this message.
  - For ftm$_t_ack_ready_receive, a message is written into the trace log stating that ftm should never receive this message.
  - For ftm$_t_ack_request, an ftm$_t_ack_ready_receive message is sent to the sw_header.source_address (presumably *FDBR*), indicating that FTM is ready to receive the recovery data for the requested time frame.
  - For ftm$_t_recovery_resend, a message is written into the trace log stating that ftm should never receive this message.
  - For ftm$_t_recovery_status, a case statement is used to determine the recovery status, as follows:
    - A status of **recovery_stat_ok** is ignored.
    - For recovery_stat_complete status, a subroutine *recovery_complete* is called to update the **recovery_record** for a completed recovery.
    - For recovery_stat_queued status, a subroutine *recovery_queued* is called to update the **recovery_record** for a recovery queued by the *FDBR*.
    - For recovery_stat_initiated status, a subroutine *recovery_init* is called to update the **recovery_record** for a recovery started by *FDBR*.
    - For recovery_stat_bad_request, recovery_stat_spanfault, recovery_stat_fatal1, recovery_stat_fatal2, a subroutine *recovery_error* is called to handle the respective error in the recovery request.

- *Recovery_Start* is given a start time and a recovery reason. The reason is checked against the following recovery_reason boundaries (0...13):
  - If the current_state indicates that a recovery has been requested and has yet to be processed, then the routine is aborted. Only one recovery may be handled at a time. The current_reason is set to the given reason.

- If an ongoing recovery has been unsuccessfully completed for an hour period, then the recovery is aborted. The current_start_time and current_time_range[1] are set to the current_time global. The current_state is set to recovery_req_sent. The net$_send_message_addr_list call is used to send an ftm$_t_full_recovery message to the FDBR. The number_recovery_attempts field is incremented, and a status message is sent to the trace log and to stats clients.

Error Conditions and Handling

In response to system call failures, recovery message information and error messages are written to the trace log. Specific error messages are listed in Table 18-21.

18.1.13 The Report Module

Purpose

Generate reports in response to list requests for specific flight data, write them into a file, and send the file to the requestor's address. This module is shared with the *FTM_Coprocess* process. If there is no *FTM-Coproc*, *FTM* handles the list request; otherwise, all requests are forwarded to *FTM-Coproc*. For this reason, the **report.ins** include file is used to hold the routine declarations for the *Report* module. See Table 18-4.

Input

Inputs are the specific flight data requests. See Table 18-2.

Output

Response files are forwarded to the requestor of the information. The **report_t** structure is used for most of these reports.

Table 18–4. Report_t Data Structure

report_t				
Library Name: etms_lib		**Purpose:** Contain user request report information.		
Element Name: ftm_user_report.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
sequence	List request sequence number			char 4
command	User command requested			char10
size	Size of data contained in report			integer16
data	The report			char4000

Routines. The *Report* module is composed of the following routines:

- *Adrfillit_Req* creates a file under the name **/sio_files/ftmtdb.mmddhhmmss**, and expands the *ASD/ADR* report with flight data.
- *Airport_Header_Get_List* is given an airport, a **start_time**, **stop_time**, and boolean arrival and returns a flight count representing the number of flights arriving (if arrival is **TRUE**) or departing from the airport within the interval provided.
- *Airportheader_Req* handles the **AIRPORTHEADER** command. It writes to the output buffer the following information: the number of airports, when the first interval started, when the last interval starts, and the c urrent time.
- *Arr_Req* handles the **ARR** command. This routine traverses the entire database and lists all arrivals at a specified airport to an output file named **ftmout.**<timestamp>. The name of the file is added to the output buffer. This is a time consuming function that is useful for diagnostic purposes.
- *Arrt_Req* handles the **ARRT** command. This routine writes the arrivals for a specified airport to an output file named **ftmout.**<time stamp>. The name of the file is added to the output buffer. It uses the airport tables and time intervals. It is not time consuming.
- *Compare_Bytes* compares the bytes of two given structures.
- *Create_Output_File* creates a report file under the name **/sio_files/ftmout.mmddhhmmss**.
- *Dep_Req* handles the **DEP** command. This routine traverses the entire database and lists all departures at a specified airport to an output file named **ftmout.**<time stamp>. The name of the file is added to the output buffer. This is a time consuming function that is useful for diagnostic purposes.
- *Dept_Req* handles the **DEPT** command. This routine writes the departures for a specified airport to an output file named **/sio_files/ftmout.**<time stamp>. The name of the file is added to the output buffer. This routine uses the airport tables and time intervals. It is not time consuming.
- *Display_Rpt_Error* writes error messages to **stdout** during report generation.
- *Dumpactive_Req* handles the **DUMPACTIVE** command. This routine dumps the flight records for all flights in the active table into a file called **active.**<time stamp>. The file name is added to the output buffer.
- *Find_Airport_In_Pacing* returns the slot in the table of airport codes that matches the code provided. If the code provided begins with a **K** which symbolizes international flights, the **K** is removed before a search is made.
- *Find_Flight_On_Index_Rpt* takes an aircraft identifier (**acid**) and a **flight_index**, calls *Hash_Acid*, and traverses the linked list looking for a match

for a flight that has not yet landed. If no match is found, then the **flight_ptr** is returned as **net_nil**, otherwise **flight_ptr** points to the matching entry.

- *Flights_Req* handles the **FLIGHTS** command. This routine traverses the entire database and lists all flights whose **acid**s begin with a specified three letters. For example if TWA is specified, then all flights for TWA will be written to the output file named **ftmout.**<time stamp>. Then, the file name is added to the output buffer. This is a time consuming routine that is useful for diagnostic purposes.
- *Get_Word* reads a buffer at a given starting point, using the space character as a delimiter, and returns the first word found and the **word_size**.
- *Hash_Acid* is a function which hashes an **acid** to obtain an index into the flight table.
- *Lifp_Req* creates a file under the name **/sio_files/ftmlifp.mmddhhmmss** and writes to it the legs of the provided flight in **rbuffer.data**.
- *List_Req* handles the **ARRIVALS**, **BOTH,** and **DEPARTURES** commands. This routine writes the data to an output file named **airport.**<time stamp> which contains the flight records for a specified set of airports and a specified time interval. The routine requires a command for up to ten airports, a **start_time**, and a **stop_time**. The name of the file is added to the output buffer.
- *Lower_Case* takes a string and converts it to all lower case.
- *Move_Bytes* takes an input buffer and size and copies it to an output buffer.
- *Output_Flight* takes a **flight_ptr** and a **stream_id**. It copies the data from **flight_array1** and **flight_array2**, pointed to by **flight_ptr**, into a local variable **out_rec**. This flight table structure is written into the given stream.
- *Tdbfillit_Req* handles the **TDBFILLIT** command. This routine is given a file name. It creates a new file **ftmtdb.**<time stamp>, which will contain the control information found in the first file combined with the flight data from *FTM* database. When complete, both file names are sent to the requesting process.
- *Validate_Req* handles the **VALIDATE** command. This is a diagnostic routine. It should not be used on an operational system, since it causes a tremendous amount of paging, which can seriously impact system efficiency. It creates its own bitmap and traverses all the **in use** tables. For each entry, it validates the route information and sets the bit in its bitmap. When complete, it compares the bitmap it made with the database bitmap and writes a report on the differences, if any, to the output buffer.

Error Conditions and Handling

Display_Rpt_Error writes error messages to the output window.

18.2 The FTM_Coprocess Process

Purpose

The purpose of this process is to handle the report requests for the *Flight Table Manager* (*FTM)* function. If this process is not running, the *FTM* process will respond to these requests.

The *FTM_Coproc* registers to an *FTM* on its own site as a report client. Once registered, it is then forwarded all flight database queries assigned to the *FTM*. The query response is returned to *FTM*, which then forwards it to the original requestor.

Execution Control

The *FTM_Coprocess* is normally started by the utility *Nodescan* whenever *Nodescan* detects that an *FTM_Coprocess* is not running.

Input

Ftm_Coprocess requires no mandatory parameters. The optional parameter **-path <etms_path>** may be supplied. In addition, it receives ETMS messages containing commands and arguments.

Output

The output is report files responding to user requests. See Table 18-4.

Processing

Upon startup, *FTM_Coprocess* loads in the flight database and registers to the *FTM* with the client type **rept**. Next, it executes the *Process* routine, which contains an endless loop. Inside this loop the mailbox is read, and the message is examined. First, the return address is placed in the output buffer. The command is extracted and a routine is called to process it. Usually these subroutines write data to a file in the **/sio_files** directory and add the file name to the output buffer. All these file names end with a time stamp, which is made up of the month, day, hour, minute, and seconds (**mmddhhmmss**). The *Process* routine adds a **linefeed** and a message stating the number of seconds it took to process the request. Finally, the output buffer is written to the mailbox.

The modules making up *FTM_Coprocess* are *Report*, shared by the *FTM*, *FTM_Coproc*, and *FTM_Coproc_Main*.

FTM_Coproc responds to *Net.Mail* statistics levels S0 through S6, where S0 and S1 requests return the same information. The S0 and S1 statistics provide overall request and message counts. S2 statistics display currently queued request jobs. S3 statistics provide detailed information about a specified queued job. S4 statistics list the most recently completed requests. S5 statistics provide detailed information about a specified completed request.

Error Conditions and Handling

Fatal errors are handled using the Aegis *Pfm_$Cleanup* system call. When such an error occurs, the cleanup handler calls the *Error_$Print_Name,* the *Flight_Table_Unmap* routine (*Ms* module), and the *Pfm_$Rls_Cleanup*.

Flight_Table_Unmap calls *Display_Error* (*Misc* module) if it has any trouble unmapping any of the checkpoint files.

Flight_Table_Create calls *Display_Error* (*Misc* module), *Flight_Table_Unmap* and *Pfm_$Signal* if it cannot map any of the checkpoint files.

Process sends back a message if it receives an invalid command. If this routine gets an error writing to the mailbox, it calls *Error_$Print* and *Pfm_$Signal*, which terminates the process.

18.3 FTM Source Code Organization

Flight Table Manager is built by a **build ftm** command to DSEE. For the sake of example, the following shell script is provided that performs the same function.

```
von
abtsev – m
rdym – on
pas  ftm_airport_table     -opt 2  -dbs   -cpu mathlib_sr10 –b    ftm_airport_table.mathlib_sr10
pas  ftm_extract           -opt2   -dbs   -cps mathlib_sr10 -b    ftm_extract.mathlib_sr10
pas  ftm_flight_handler    -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_flight_handler.mathlib_sr10
pas  ftm_interface         -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_interface.mathlib_sr10
pas  ftm_main              -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_main.mathlib_sr10
pas  ftm_misc              -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_misc.mathlib_sr10
pas  ftm_ms                -opt 2  -dsb   -cpu mathlib_sr10 -b    ftm_ms.mathlib_sr10
pas  ftm_pacing            -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_pacing.mathlib_sr10
pas  ftm_parse             -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_parse.mathlib_sr10
pas  ftm_raw               -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_raw.mathlib_sr10
pas  ftm_recovery          -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_recovery.mathlib_sr10
pas  ftm_report            -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_report.mathlib_sr10
pas  ftm_coproc            -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_coproc.mathlib_sr10
pas  ftm_coproc_main       -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_coproc_main.mathlib_sr10
ftn  ftm_great_circle      -opt 2  -dbs   -cpu mathlib_sr10 -b    ftm_great_circle.mathlib.sr_10
```

Once the files have been compiled, the following command must be issued in order to bind the files together into two executable programs:

```
von
rdym – on
bind <<!
ftm_airport_table.mathlib_sr10.bin
ftm_extract.mathlib_sr10.bin
ftm_flight_handler.mathlib_sr10.bin
ftm_interface.mathlib_sr10
fim_main.mathlib_sr10.bin
ftm_misc.mathlib_sr10.bin
ftm_ms.mathlib_sr10.bin
```

```
ftm_pacing.mathlib_sr10.bin
ftm_parse.mathlib_sr10.bin
ftm_raw.mathlib_sr10.bin
ftm_recovery.mathlib_sr10.bin
ftm_report.mathlib_sr10.bin
ftm_great_circle.mathlib_sr10.bin
-b ftm
!
bind <<!
ftm_coproc.mathlib_sr10.bin
ftm_coproc_main.mathlib_sr10.bin
ftm_report.mathlib_sr10.bin
-b ftm_coproc
!
```

18.4 FTM Data Structures

The *FTM* uses a variety of data structures. This section discusses the data structures used in the FTM database and three others used throughout the function.

18.4.1 FTM Database Data Structures

Tables 18-5 through 18-18 contain information on the format of the keys to the *Shared Region* files. These files contain all the information about active and pending flights for the *FTM*.

Flight_array1 is the key to **ftm_checkpoint_file_a** and is defined by the following:

Flight_array1 is of type **flight_array1_ptr.**

This type is defined as ^**flight_array1_t. Flight_array1_t** is an array of **flight_table_entry1_t.**

Table 18–5. flight_array1_t Data Structure

flight_array1_t				
Library Name: etms_lib		**Purpose:** Array of records of **flight_table_entry_t**		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_array1_t	Array of flight_table_entry1_t.	total_flight_records = 125,000	1…total_flight_ records	array

Table 18–6. flight_table_entry1_t Data Structure

flight_table_entry1_t				
Library Name: etms_lib		**Purpose:** To contain the most often used portion of the flight record.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_index	Flight index			integer32
deletion_time	Time to remove flight from database	Whole seconds		integer32
next_one	Position in linked list			integer32
acid	Aircraft identifier			char7
dep_air	Departure airport			char5
arr_air	Arrival airport			char5
active	Flight's active status	See 18.4.1.1		char
update_type	Type of last message received on flight	See 18.4.1.2		char
filler	Reserved			char

18.4.1.1 Active Flights

The active field in **flight_table_entry1_t** represents the current state of the aircraft and can have the values shown in the text that follows.

Value for Flights That Have Not Yet Flown

P – A flight that has received an FZ, FS, or AF message, and is waiting for a position report.

Values for Flights in the Air

A - An active flight that has received one of the following messages within the last seven minutes:

TZ, DZ, TO, TA, or FZ

E - An expired flight (ghost) whose last position update is more than seven minutes old.

Values for Flights That Have Landed

B - A deactivated flight due to receipt of an SI message (substitution or cancellation).

C - A deactivated flight due to receipt of a Control Cancel message.

1 - A deactivated flight due to 1) receipt of an AZ message, or 2) receipt of either a TTM_FTM or CRITICAL message with a flight status indicating that the flight has completed.

2 - A deactivated flight due to receipt of an RZ message.

3 - An international flight that is deactivated because of meeting one of the following conditions:

- The flight is adjacent to its destination airport.
- The flight distance traveled is greater than the distance to its destination airport.
- The flight is moving away from the airport, and the ghost to waypoint flag is false.

4 - A flight deactivated because it has been in a holding pattern for more than one hour.

5 - A flight deactivated because it is adjacent to its destination airport while moving away from the airport.

6 - A ghosting flight deactivated because of one of the following two reasons:

- It had a speed of less than 10 knots.
- It had no valid waypoint to ghost toward and its destination is unknown.

7 - A flight deactivated because its position update time is unknown.

8 - A flight deactivated due to receipt of an RY message.

9 - A flight deactivated due to receipt of an RS message.

o - A deactivated flight due to an adjacency to airport test. This occurs when the distance traveled is greater than the distance to the destination, as computed for the previous map file, and the aircraft is moving away from the airport.

d - A flight deactivated because its computed distance travelled places it within 5 nautical miles of the destination ariport.

g - A flight deactivated due to meeting the following conditions:

- It is adjacent to its destination airport OR its distance traveled is greater than the distance to its destination airport.
- It is moving away from the airport and the ghost to the waypoint flag is true.

Value for Flights Flagged for Removal from the FTM Database

- **X** - A flight that is no longer active. This flight has been in the database for its life period or has been cancelled. The flight is due to be physically removed from the database during the next purge cycle.

18.4.1.2 Update Type

The **Update Type** field can have one of the following values:

- **A** = Last update was an AF message
- **B** = Last update was a 5-MINUTE SETBACK message
- **C** = Last update was an RY message
- **D** = Last update was a DZ message
- **E** = Last update was an EDCT message
- **F** = Last update was an FZ message
- **G** = Last update was a RAW TZ
- **H** = Last update was a CONTROL CANCEL message
- **I** = Last update was a CRITICAL message
- **J** = Last update was a RAW DZ
- **K** = Last update was a BLOCK ALTITUDE message
- **L** = Last update was an AZ message
- **M** = Last update was a TTM_FTM message
- = Last update was a TO message
- **R** = Last update was an RS message
- **S** = Last update was an FS message
- **T** = Last update was a TZ message
- **U** = Last update was a UZ message
- **W** = Last update was a TA message
- **X** = Last update was an SI CANCEL message

- **Y** = Last update was an FY message
- **Z** = Last update was an RZ message

Flight_array2 is the key to **ftm_checkpoint_file_g** and is defined by the following:

Flight_array2 is of type **flight_array2_ptr.**

This type is defined as ^**flight_array2_t. Flight_array2_t** is an array of **flight_table_entry2_t.**

Table 18–7. flight_array2_t Data Structure

flight_array2_t				
Library Name: etms_lib		**Purpose:** Array of **flight_table_entry2_t**		
Element Name: ftm.mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_array2_t	Array of flight_table_entry2_t.	total_flight_records = 125,000	1…total_flight_ records	array

Table 18–8. flight_table_entry2_t Data Structure

flight_table_entry2_t				
Library Name: etms_lib		**Purpose:** To contain the fixed portion of the flight record.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
filed_ascii_alt	Filed ASCII altitude			char8
ascii_altitude	ASCII altitude			char8
arrival_fix	Arrival fix			char6
ac_type	Aircraft type			char4
dep_schedule	Scheduled departure time	Whole seconds		integer32
dep_proposed	Proposed departure time	Whole seconds		integer32
dep_estimate	Estimated departure time	Whole seconds		integer32
dep_controlled	Controlled departure time	Whole seconds		integer32
dep_actual	Actual departure time	Whole seconds		integer32
ogtd	Original gate time of departure	Whole seconds		integer32
arr_schedule	Scheduled arrival time	Whole seconds		integer32
arr_proposed	Proposed arrival time	Whole seconds		integer32
arr_estimate	Estimated arrival time	Whole seconds		integer32
arr_controlled	Controlled arrival time	Whole seconds		integer32
arr_actual	Actual arrival time	Whole seconds		integer32
ogta	Original gate time of arrival	Whole seconds		integer32
last_distance	Most recently computed distance to destination	Miles		integer32
arrival_fix_time	Arrival fix time	Whole seconds		integer32
last_posit	Time of last position update	Whole seconds		integer32
posit_2_time	Time at position 2	Whole seconds		integer32
posit_3_time	Time at position 3	Whole seconds		integer32
last_upd_time	Last time received message for flight	Whole seconds		integer32
creation_time	Time flight created by FTM	Whole seconds		integer32
route_ptr	Route pointer			integer32
calc_time	Time at calculated position	Whole seconds		integer32
route_ptr_size	Size of Field 10 (flight's route)			integer32
route_size_code	Number of bytes required to store route in FTM database			integer32
ete	Estimated time enroute	Minutes		integer16

Table 18–8. flight_table_entry2_t Data Structure (continued)

flight_table_entry2_t				
Library Name: etms_lib		**Purpose:** To contain the fixed portion of the flight record.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
altitude	Altitude	100 Feet		integer16
altitude2	Altitude 2 (for use with block altitudes)	100 Feet		integer16
prev_lat	Previous latitude	Degrees*60 + Minutes		integer16
prev_lon	Previous longitude	Degrees*60 + Minutes		integer16
cur_lat	Current latitude	Degrees*60 + Minutes		integer16
cur_lon	Current longitude	Degrees*60 + Minutes		integer16
next_lat	Next latitude	Degrees*60 + Minutes		integer16
next_lon	Next longitude	Degrees*60 + Minutes		integer16
second_lat	Second latitude	Degrees*60 + Minutes		integer16
second_lon	Second longitude	Degrees*60 + Minutes		integer16
third_lat	Third latitude	Degrees*60 + Minutes		integer16
third_lon	Third longitude	Degrees*60 + Minutes		integer16
groundspeed	Speed	Knots		integer16
dest_lat	Destination latitude	Degrees*60 + Minutes		integer16
dest_lon	Destination longitude	Degrees*60 + Minutes		integer16
filed_alt	Filed altitude	100 Feet		integer16
filed_speed	Filed speed	Knots		integer16
filed_alt2	Filed altitude 2 (for use with block altitude)	100 Feet		integer16
could_have_landed	Whether the flight projected to have landed by FTM	100 Feet	0=no 1=yes	integer16
calc_lat	Calculated latitude	Degrees*60 + Minutes		integer16
calc_lon	Calculated longitude	Degrees*60 + Minutes		integer16

Table 18–8. flight_table_entry2_t Data Structure (continued)

flight_table_entry2_t				
Library Name: etms_lib		**Purpose:** To contain the fixed portion of the flight record.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
filler	Reserved			array[1…16] of integer16
flags	Flight flags	See 18.4.1.3		integer16
source_flags	Flight flags – part 2	See 18.4.1.4		integer16
remarks_flags	Field 11 comment flags			integer16
geo_filter	Reserved for future geographical filter bitmask			integer16
altitude_type	Altitude type			char
update_ctr	ARTCC from which last update originated			char
flight_status	Status of flight	See 18.4.1.5		char
air_category	Aircraft category (physical class)	See 18.4.1.6		char
user_category	User category	See 18.4.1.7		char
wght_category	Weight category	S=Small, L=Large, H=Heavy	S,L,H	char
arrival_ctr	Arrival ARTCC			char
depart_ctr	Departure ARTCC			char
waypoints	Number of 4 byte waypoints in flight's route			char
sectors	Number of 6 byte waypoints in flight's route			char
fixes	Number of 6 byte fixes in flight's route			char
airways	Number of 6-byte airways in flight's route			char
centers	Number of 3-byte centers in flight's route			char

Table 18–8. flight_table_entry2_t Data Structure (continued)

flight_table_entry2_t				
Library Name: etms_lib		**Purpose:** To contain the fixed portion of the flight record.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
field_10_size	Size of field 10 (flight's route)			char
time_used	Time field indicator			char
prefix_digit	Field 3 aircraft type prefix digit			char
prefix_char	Field 3 aircraft type prefix char			char
suffix_char	Field 3 aircraft type suffix char			char
international	Whether flight is international		TRUE,FALSE	boolean
ghost_to_route	Whether flight has a next position to ghost toward		TRUE,FALSE	boolean

18.4.1.3 Flags

The **Flags** field can have any of the following bit values:

0 = Military flight

1 = Weight Category 1 (Small) of aircraft

2 = Weight Category 2 (Large) of aircraft

3 = Weight Category 3 (Heavy) of aircraft

4 = Received CANCEL message

5 = Received CRITICAL message

6 = Received POSITION message

7 = Received ROUTE message

8 = Received TIME message

9 = Received TTM_FTM message

10 = Received TZ message

11 = Received BLOCK_ALT message

12 = Received RAW TZ message

13 = Received RAW DZ message

14 = Reserved

15 = Flight in holding pattern

18.4.1.4 Source Flags

The **Source_Flags** field can have any of the following bit values:

0 = Received DZ message

1 = Received FZ message

2 = Received UZ message

3 = Received AF message

4 = Received FS message

5 = Received AZ message

6 = Received RS message

7 = Received RZ message

8 = Received TO message

9 = Received TA message

10 = Received FY message

11 = Received RY message

12 = Received EDCT message

13 = Received 5-SETBACK message

14 = Received SI CANCEL message

15 = Received CONTROL CANCEL message

18.4.1.5 Flight Status

The **Flight Status** field can have one of the following values:

N = None

S = Scheduled

F = Filed

A = Active

R = Ascending

C = Cruising

D = Descending

T = Completed

X = Cancelled

E = Error

“” = Not determined

18.4.1.6 Air Category

The **Air Category** field can have one of the following values:

“” = Not determined

P = Piston

T = Turbo

J = Jet

18.4.1.7 User Category

The **User Category** field can have one of the following values:

O = Other

T = Air taxi

F = Cargo

C = Commercial

G = General aviation

M = Military

“” = Not determined

18.4.1.8 Miscellaneous parameters

Miscellaneous parameters in flight_array2 are:

waypoints – number of 4 byte binary entries, in Lat/Lon format of degrees*60+minutes

sectors – number of 6 byte ASCII entries, left justified, blank filled, its starting address is pointed to by **seek_key** + **waypoints***4

fixes – number of 6 byte ASCII entries, left justified, blank filled, its starting address is pointed to by **seek_key** + **waypoints***4 + **sectors***6

airways – number of 6 byte ASCII entries its starting address is pointed to by **seek_key** + **waypoints***4 + **sectors***6+**fixes***6

centers – number of 3 byte ASCII entries its starting address is pointed to by **seek_key** + **waypoints***4 + **sectors***6 + **fixes***6 + **airways***6

route_size – number of ASCII bytes, starting address is pointed to by **seek_key** + **waypoints***4 + **sectors***6+**fixes***6 + **airways***6 + **centers***3

total_ptr_size – waypoints*4 + **sectors***6+**fixes***6 + **airways***6 + **centers***3 + **route_size**

The **arrival_time_code** or **departure_time_code** has the following values:

0 = Not specified

1 = Actual

2 = Traffic Model Functions/estimated

3 = Controlled

4 = Proposed

5 = Scheduled

The following is used to determine **time_used**:

time_used = **arrival_time_code** * 20 + **departure_time_code**

The following is used for a flight using actual departure and proposed arrival:

time_used := chr (1 + 20 * 4) = chr (81)

Flight_array_rte is the key to **ftm_checkpoint_file_b** and is defined by the following:

Flight_array_rte is of type **flight_rte_array_ptr.**

This type is defined as ^**flight_rte_array_t. Flight_rte_array_t** (see Table 18-9) is an array of **flight_table_rte_entry_t** (see Table 18-10)**.**

Table 18–9. flight_rte_array_t Data Structure

flight_rte_array_t				
Library Name: etms_lib		**Purpose:** Array of **flight_table_rte_entry_t**		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_rte_array_t	Array of flight_table_rte_entry_t	= total_flight_records*8	1…total_rte_pages	array

Table 18–10. flight_rte_entry_t Data Structure

flight_rte_entry_t				
Library Name: etms_lib		**Purpose:** To contain the route portion of the flight record		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
text	Text in this page		1…32	array of char

Flight_table_hash is the key to **ftm_checkpoint_file_c** and is defined by the following:

Flight_table_hash is of type **flight_table_hash_ptr.**

This type is defined as ^**flight_table_hash_t**. **Flight_table_hash_t** is an array of integer32.

Table 18–11. flight_table_hash_t Data Structure

flight_table_hash_t				
Library Name: etms_lib		**Purpose:** To contain the hashing table for the flight database		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flight_table_hash_t	Array of integer32.	flight_table_hash _size= 20001	1…flight_table_hash _size	array

Flight_storage is the key to **ftm_checkpoint_file_d** and is defined by the following:

Flight_storage is of type ^**flight_storage_bit_t.**

Flight_storage_bit_t is a record. Two elements of this record are of type **zero_one**.

Table 18–12. flight_storage_bit_t Data Structure

flight_storage_bit_t				
Library Name: etms_lib		**Purpose:** To contain the flight storage information		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
flights	Array of zero_one_t. Zero_one_t = 0.1	total_flight_records = 125,000	1…total_flight_ records	array
routes	Array of zero_one_t. Zero_one_t = 0.1	total_rte_pages 125,ooo*8	1…total_rte_pages	array

Flight_active is the key to **ftm_checkpoint_file_e** and is defined by the following:

Flight_active is of type **active_table_ptr_t.**

This type is defined as ^**active_table_t. Active_table_t** is an array of **active_entry_t**.

Table 18–12. active_table_t Data Structure

active_table_t				
Library Name: etms_lib		**Purpose:** To contain the active flight information		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
bitmap	Array of zero_one_t. Zero_one_t = 0.1	number_actives… = 6,000	1…number_actives_ allowed	array
hash	Hash table for active flights	active_flight_ hash_size=1,001	1…active_flight_ hash_size	array of integer
entry	Array of active_entry_t		1..number_actives_ allowed	array

Table 18–14. active_entry_t Data Structure

active_entry_t				
Library Name: etms_lib		**Purpose:** To contain the active flight entries		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
acid	Aircraft identifier.			char7
flight	Link to flight entry in flight array.			integer32
next_one	Link for next flight in linked list.			integer
index	Flight index.			integer32
filler	Reserved.			char3

Flight_airport is the key to **ftm_checkpoint_file_f** and is defined by the following:

Flight_airport is of type **airport_space_ptr_t.**

This type is defined as ^**airport_space_t.** **Airport_space_t** has an element composed of **airport_entry_t**. **Airport_entry_t** is an array of **airport_time_t**.

Table 18–15. airport_space_t Data Structure

airport_space_t				
Library Name: etms_lib		**Purpose:** To contain the airport information		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
airports_used	Number of airports used.			integer
interval_start	Interval start time.			integer32
which_is_first	First airport name.			integer
name	Airport name.	max_number_ airport=100	1…max_number_ airports	array of char4
entry	Array of airport_entry_t.		1…max_number_ airports	array

Table 18–16. airport_entry_t Data Structure

airport_entry_t				
Library Name: etms_lib		**Purpose:** Array of **airport_time_t**		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
airport_entry_t	Array of airport_time_t.	† See below.	1…airport_number _intervals	array

† Airport_number_intervals = (retention_period div [60*60]*4*2*4)

Table 18–17. airport_time_t Data Structure

airport_time_t				
Library Name: etms_lib		**Purpose:** To contain airport time information		
Element Name: ftm_mmu.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
arr_count	Arrival count.			integer
dep_count	Departure count.			integer
arr_list	Arrival list.	max_flights_in_ airport_list = 50	1…max_flights_in_ airport_list	array of integer32
dep_list	Departure list.		1…max_flights_in_ airport_list	array of integer32

Table 18–18. shared_region_header_t Data Structure

shared_region_header_t				
Library Name: etms_lib		**Purpose:** To contain information about the shared regions		
Element Name: ftm_constants.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
creation_time	Time of shared region creation.			time_$clock_t
running_time	Time elapsed since region creation.			integer32
start_count	Number FTM starts since region creation.			integer32
stop_count	Number FTM shut downs since region creation.			integer32
total_flights_made	Flights added to database since region creation.			integer32
total_flights_del	Flights removed from database since region creation.			integer32
shut_down_time	Last time FTM was shut down.			integer32

18.4.2 The flight_table_entry_t Data Structure

Table 18-19 contains information on the format of the **flight_table_entry_t** record structure. This record is the combination of the data in **flight_array1** and **flight_array2** described in the preceding section. This structure is traversed each time a **map** file is produced.

Table 18–19. flight_table_entry_t Data Structure

flight_table_entry_t				
Library Name: etms_lib		**Purpose:** To contain the flight database entries for FTM.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
acid	Aircraft identifier			char10
ascii_altitude	ASCII altitude			char8
ascii_file_alt	Filed ASCII altitude			char8
arrival_fix	Arrival fix			char6
dep_air	Departure airport			char5
arr_air	Arrival airport			char5
ac_type	Aircraft type			char4
flight_index	Flight index			integer32
dep_schedule	Scheduled departure time	Whole seconds		integer32
dep_proposed	Proposed departure time	Whole seconds		integer32
dep_estimate	Estimated departure time	Whole seconds		integer32
dep_controlled	Controlled departure time	Whole seconds		integer32
dep_actual	Actual departure time	Whole seconds		integer32
ogtd	Original gate time of departure	Whole seconds		integer32
arrival_fix_time	Arrival fix time	Whole seconds		integer32
arr_schedule	Scheduled arrival time	Whole seconds		integer32
arr_proposed	Proposed arrival time	Whole seconds		integer32
arr_estimate	Estimated arrival time	Whole seconds		integer32
arr_controlled	Controlled arrival time	Whole seconds		integer32
arr_actual	Actual arrival time	Whole seconds		integer32
ogta	Original gate time of arrival	Whole seconds		integer32
route_ptr	Route pointer			integer32
calc_time	Time flight at calculated position	Whole seconds		integer32
ete	Estimated time enroute	Minutes		integer16
altitude	Altitude	100 feet		integer16
altitude2	Altitude2 (for use with block altitudes)	100 feet		integer16
altitude_filed	Filed altitude	100 feet		integer16

Table 18–19. flight_table_entry_t Data Structure (Continued)

flight_table_entry_t				
Library Name: etms_lib		**Purpose:** To contain the flight database entries for FTM.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
cur_lon	Current longitude	Degrees*60 + Minutes		integer16
cur_lat	Current latitude	Degrees*60 + Minutes		integer16
calc_lon	Calculated longitude	Degrees*60 + Minutes		integer16
calc_lat	Calculated latitude	Degrees*60 + Minutes		integer16
groundspeed	Speed	Knots		integer16
speed_filed	Filed speed	Knots		integer16
dest_lat	Destination latitude	Degrees*60 + Minutes		integer16
dest_lon	Destination longitude	Degrees*60 + Minutes		integer16
route_ptr_size	Size of field 10 (flight's route)			integer16
flags	Flight flags	See flight table_ entry1_t		integer16
source_flags	Flight flags – part 2	See flight table_ entry1_t		integer16
remarks_flags	Field 11 comments flags			integer16
geo_filter	Reserved for future geographical bitmask			integer16
filler	Reserved			array[1…16] of integer16
altitude_type	Altitude type			char
active	Flight's active status	See flight table_ entry1_t		char
update_type	Message type of last update	See flight table_ entry1_t		char
flight_status	Status of flight	See flight table_ entry1_t		char
air_category	Aircraft category (physical class)	See flight table_ entry1_t		char
user_category	User category	See flight table_ entry1_t		char
arrival_ctr	Arrival center			char

Table 18–19. flight_table_entry_t Data Structure (Continued)

flight_table_entry_t				
Library Name: etms_lib		**Purpose:** To contain the flight database entries for FTM.		
Element Name: flight_table.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
departure_ctr	Departure center			char
waypoints	Number of 4-byte waypoints in flight's route			char
sectors	Number of 6-byte sectors in flight's route			char
fixes	Number of 6-byte fixes in flight's route			char
airways	Number of 6-byte airway identifiers in flight's route			char
centers	Number of 3-byte ARTCC identifiers in flight's route			char
field_10_size	Size of field 10 (flight's route)			char
time_used	Time field indicator			char
prefix_digit	Field 3 aircraft type prefix digit			char
prefix_char	Field 3 aircraft type prefix char			char
suffix_char	Field 3 aircraft type suffix char			char

Refer to the previous section for more details on the contents of the fields of the **flight_table_entry_t** record structure.

18.4.3 The map_output_record_t Data Structure

The reports sent to the *ASD* by *FTM* are known collectively as **map** files. The **map** file currently consists of two files: the **map** flight file and the **map** route file. The flight file contains information about each flight in a fixed record format. The route file is composed of variable sized records which are the exact images of the structures pointed to by the **route_ptr** field of **flight_table_entry_t**. This structure, **map_output_record_t**, is written into the **map** file for each flight. Table 18-20 contains the format of the **map_output_record_t** record structure.

Refer to the *flight_table_entry2_t* section for details on the values of the flags field.

The following provides a more detailed description of some fields in the record:

x_current, **y_current –** These fields define the estimated aircraft position based upon the last two reported positions to the *FTM*. These values are time-adjusted by aircraft speed from the last reported position. If the flight is ghostable and no TZs have been received, the position estimate is a straight line from the last reported position to the destination airport. For an international flight under these same circumstances, a great circle route between last reported position and destination is made.

x_previous, **y_previous –** These fields are similar to the above except for using the second most recent reported position.

heading – This field is calculated from the previous and current flight data.

symbol – This field states how the aircraft should be displayed:

^ **–** Display as a circle

a through **h –** indicates each angle from 0-359 in units of 45 degrees

i through **p –** similar to a through h except the aircraft is a ghost

@ – Indicates the flight is a time record

seek_key – This field contains the address in the **map** route file of the route data for this flight. It is set to **16#FFFFFFFF** if there are no route data.

The **X**, **Y** values in the records are in Albers projections.

Table 18–20. map_output_record_t Data Structure

map_output_record_t				
Library Name: etms_lib		**Purpose:** To contain the flight information used to build MAP files.		
Element Name: maps_interface.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
acid	Aircraft identifier			char10
ascii_altitude	ASCII altitude			char8
origin	Departure airport			char5
destination	Arrival airport			char5
ac_type	Aircraft type			char4
eta	Estimated time of arrival			integer32
seek_key	Key to route data for flight			integer32
altitude	Altitude	100 feet		integer16
altitude2	Altitude2 (for use with block altitudes)	100 feet		integer16
current_x	Alber's current longitude			integer16
current_y	Alber's current latitude			integer16
previous _x	Alber's previous longitude			integer16
previous _y	Alber's previous latitude			integer16
current_lat	Current latitude	Degrees*60 + Minutes		integer16
cur_lon	Current longitude	Degrees*60 + Minutes		integer16
previous _lat	Previous latitude	Degrees*60 + Minutes		integer16
previous _lon	Previous longitude	Degrees*60 + Minutes		integer16
heading	Heading			integer16
groundspeed	Speed	Knots		integer16
cta	Cleared time to arrival	Minutes		integer16
flags	Flight flags	See flight_table_ entry1_t		integer16
source_flags	Flight flags – part 2	See flight_table_ entry1_t		integer16
remarks_flags	Field 11 comments flags			integer16
geo_filter	Reserved for future geographical bitmask			integer16

Table 18–20. map_output_record_t Data Structure (continued)

map_output_record_t				
Library Name: etms_lib		**Purpose:** To contain the flight information used to build MAP files.		
Element Name: maps_interface.ins				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
filler	Reserved			array (1…13) integer16
center_id	ARTCC identifier			char
altitude_type	Altitude type			char
lat_lon_heading	Lat/lon heading divided by 2			char
symbol	Indicator for flight display type			char
waypoints	Number of 4-byte waypoints in flight's route			char
sectors	Number of 6-byte sectors in flight's route			char
fixes	Number of 6-byte fixes in flight's route			char
airways	Number of 3-byte airways in flight's route			char
centers	Number of 3-byte center IDs in flight's route			char
field_10_size	Size of field 10 (flight's route)			char
actr	Arrival ARTCC			char
dctr	Departure ARTCC			char
last_update	Last update message type	See flight_table_ entry1_t		char
air_cat	Aircraft category	See flight_table_ entry1_t		char
prefix_digit	Field 3 aircraft type prefix digit			char
prefix_char	Field 3 aircraft type prefix char			char
suffix_char	Field 3 aircraft type suffix char			char
ghost_to_rte	Whether flight has a next position to ghost toward		TRUE,FALSE	boolean

Table 18–21. Error Messages

#	Error Message	Description	Responsible Module
1	...shutting down...argument does not exist *[NOTE: This error message is written to the screen; all others are written to the trace file.]*	FTM was invoked without a valid configuration filename specified.	Main
2	bad pacing airport line	Invalid line in airport location file.	Airport Table
3	invalid route size	Flight has invalid route information.	Extract
4	cannot open route file	FTM unable to create route file.	Extract
5	cannot open map file	FTM unable to create map file.	Extract
6	error writing to route file	FTM unable to write to route file.	Extract
7	error writing to map file	FTM unable to write to map file.	Extract
8	invalid hash value	FTM database access error.	Flight–Handle
9	received bad registration notice from <ETMS address>	FTM failed in attempt to register as transaction data client to FDBD or a master FTM.	Interface
10	ignoring recovery message from <ETMS address>	FTM received recovery message from site other than registered site.	Interface
11	FTM input file is not available - needed for execution	FTM unable to open airport location file: '/etms5/ftm/data/airstrip.dat'.	Airport Table, Misc.
12	invalid or non-existent keyword file	FTM unable to open Field 11 keyword file: '/etms5/shared/data/remarks_keywords'.	Interface, Misc.
13	FTM no data timeout	FTM failed to receive transaction data for over 1 minute.	Misc.
14	Switching sites because no data in 1 minute	FTM switching from primary to secondary FDBD site, or vice-versa, due to data timeout.	Misc.
15	Switching sites because no fdbd connection within 3 minutes of registration attempt	FTM switching from primary to secondary FDBD site, or vice-versa, due to registration timeout.	Misc.
16	invalid bit in keyword file	Keyword file has bad data entry.	Misc.
17	no longer attempting recovery after one hour	Recovery timeout notification.	Recovery